NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

REPRESENTATION TECHNIQUES FOR RELATIONAL LANGUAGES
AND THE WORST CASE ASYMPTOTICAL TIME COMPLEXITY
BEHAVIOUR OF THE RELATED ALGORITHMS

by

Suha Futaci

June, 1982

Thesis Advisor: Bruce J. MacLennan

Approved for public release;   distribution unlimited

T205420

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Representation Techniques for Relational Languages and the Worst Case Asymptotical Time Complexity Behaviour of the Related Algorithms		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June, 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Suha Futaci		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT. PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June, 1982
		13. NUMBER OF PAGES 352
14. MONITORING AGENCY NAME & ADDRESS(*If different from Controlling Office*)		15. SECURITY CLASS. *(of this report)*
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Complexity Relation

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This thesis is aimed at determining the worst case asymptotical time complexity behaviour of algorithms for relational operations that work on extensionally or intensionally represented binary relations. Those relational operations come from a relational language being designed at Naval Postgraduate School. One particular extensional representation technique and two intensional representation techniques are proposed. The above analysis in turn determines the feasibility of implementing a subset of the relational language on conventional architectures.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73
S/N 0102-014-6601 |

Representation Techniques for Relational Languages and
the Worst Case Asymptotical Time Complexity Behaviour
of the Related Algorithms

by

Suha Futaci
Lieutenant (Junior Grade), Turkish Navy
Turkish Naval Academy, 1976

Submitted in partial fulfillment of
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1982

ABSTRACT

This thesis is aimed at determining the worst case asymptotical time complexity behaviour of algorithms for relational operations that work on extensionally or intensionally represented binary relations. Those relational operations come from a relational language being designed at Naval Postgraduate School. One particular extensional representation technique and two intensional representation techniques are proposed. The above analysis in turn determines the feasibility of implementing a subset of the relational language on conventional architectures.

TABLE OF CONTENTS

5

I.   INTRODUCTION

This thesis is aimed at analyzing the worst case asymptotical time complexity behaviour of the algorithms associated with the relational operations that work on the extensionally or intensionally represented binary relations.

The most obvious representation of a relation is the extensional representation, in which all the elements of a relation are explicitly represented in memory. There are various kinds of extensional representation techniques that will be explained in detail later.

When the use of memory is critical so that it is uneconomical to represent relation explicitly in the memory, intensional representation techniques should be used. Here a relation or set is represented by a formula or expression for computing that relation or set. Operations on the set or relation are implemented as formal operations on the expression. Because the relations have well defined algebric properties, this seems feasible. As we can see, an intensional representation is really just a variant of a lazy evaluation mechanism. [Ref. 1] We will try to decide on the feasibility of this kind of mechanism in Section III.

The relational operations we mentioned above belong to a relational language being designed at Naval Postgraduate School. [Ref. 2] In relational programming entire relations

7

are manipulated rather than individual data. This is analogous to functional programming [Ref. 3], in which entire functions are the values manipulated by the operators. Because the set of all functions is the subset of the set of all relations, relational programming subsumes functional programming. Hence anything that can be done with functional programming can be done with relational programming. Thus relational programming has many of the advantages of functional programming. Although relations are more general than functions, their laws are often simpler. In addition, relational programming more directly supports non-linear data structures such as graphs and digraphs, than does functional programming. In relational programming the basic data values are themselves relations; on the other hand in functional programming there is a separate class of objects used for data structures.

The objective of this research is to determine the feasibility of implementing the relational language on conventional architectures by doing the worst case asymptotical time complexity analysis of the algorithms associated with the relational operations. In Section II we will focus on the algorithms that work on the extensionally represented relations and sets. In Chapter III we will inspect the intensional algorithms and define a mechanism to do the relational operations intensionally. In Chapter IV we

will focus on the issue of defining pure intensional system to see if we can do the relational operations without representing any relation or set extensionally.

A. BACKGROUND INFORMATION

1. Theorems and Definitions

In this section we will provide information on relations, and the extensional representation techniques for the relations and sets. We will state some important theorems that will be helpful in our analysis and prove them. Because our relational operations work on binary relations we will be focusing on the properties of binary relations. We will assume that the reader already has some background on relations.

We often want to treat collections of two-ary tuples where the components of each tuple are the elements of some sets. The set of all such two-ary tuples is defined as follows:

Definition 1: Let A and B be sets. The cartesian product of the sets A and B denoted by AXB, is the set of all two-ary tuples such that:

$$\{ <a1,a2> \mid (a1 \; \epsilon \; A) \; \wedge \; (a2 \; \epsilon \; B)\}$$

The following definition provides some additional terminology.

Definition 2: Let R be a binary relation over AXB. The set A is the domain or the "Right Members Set" (RIM set) of R; B is the codomain or the "Left Members Set" (LEM set) of R.

Binary relations are just another representation of graphs so the following definitions and theorems will be useful in our study.

Definition 3: A digraph is an ordered pair F=<C,R> where R is a binary relation on the set C. The set C is said to be the set of vertices of digraph F and the tuples of R corresponds to arcs (edges) of F.

Definition 4: Let F=<C,R> be a digraph with nodes d and e. An undirected path G from d to e is a finite sequence of nodes G =<b_0, b_1, b_2,....,b_n> such that:

1. b_0 = d

2. b_n = e

3. For all b_i such that n >= i >= 0, either $b_i R b_{i+1}$ or $b_{i+1} R b_i$.

If $b_i R b_{i+1}$ for all b_i, n > i > 0, then G is a directed path from d to e. The node d is the initial node of G and e is the terminal node of G. The length of the path G is n. If all the nodes of G are distinct except the first and last then G is a simple path. If b_0 is the same as b_n, then G is a cycle. If G is both simple path and a cycle, then G is a simple cycle.

Definition 5:   Let S be a binary relation on B.   Then S is reflexive if xSx for every x in B.   S is irreflexive if {not(xSx)} is true for every x in B.   S is symmetric if xSy implies ySx for every x,y ε в.   S is antisymmetric if xSy and ySx together imply x=y for every x,y ε B.   S is transitive if xSy and ySz together imply xSz for every x,y,z ε B.

Definition 6:   Let R be a binary relation on a set B. The transitive (reflexive, symmetric) closure of R is the relation S such that:

1.   S is the super set of R.

2.   S is transitive (reflexive, symmetric).

3.   For any transitive (reflexive, symmetric) relation T, if T is the super set of the R then T is the super set of the relation S.

We can denote the transitive closure of R by trans(R), the reflexive closure of R by refl(R), the symmetric closure by symm(R), and the transitive, reflexive closure by trans-refl(R).   Obtaining the closure of a binary relation can be easily understood in terms of digraphs.   For example, a digraph represents a reflexive binary relation if it has loops on every node.   So given a binary relation represented by a digraph we can obtain the reflexive closure of this relation by adding a loop to every node of the digraph which does not already have one.   Let E be the equality relation on an arbitrary set X; that is,

11

$$E = \{<a,a,>\mid a \; \epsilon \; X\}$$

then by using this relation we can state a theorem as
follows:

Theorem 1:   Let R be a binary relation on a set B.
Then refl(R)=R   E, where E is the equality relation on the
set B.

Proof: Let S = R$\cup$E.   We show that S satisfies
Definition 6.   By construction S is reflexive and S is the
super set of R.   Assume T is a reflexive relation on B and T
is the super set of R.   We have to show T is the super set of
S.   Let's take an arbitrary tuple, say <s,t>, which is the
member of R.   If s=t, then <s,t>ϵT because the T is
reflexive.   If <s,t>ϵR, then <s,t>ϵT because the T is the
super set of R by assumption.   So if <s,t>ϵS, then <s,t,>ϵT.
So as a result, the definition 6 is satisfied and S=refl(R).

Definition 7:   Let R be a binary relation on a set A
and let n be a natural number.   Then, the nth power of R,
denoted R^n, is defined as follows:

1. R^0 is the relation of equality on the set A:

$$R^0 = \{<x,x>\mid x \epsilon A\}$$

2. $R^{n+1} = R^n R$

Theorem 2:   Let R be a binary relation on the set B.
Then

$$\text{trans}(R) = \bigcup_{i=1}^{\infty} R^i = R \cup R^2 \cup R^3 \cup R^4 \; \ldots\ldots$$

Proof: The proof can be done in two parts.

1. $\bigcup_{i=1}^{\infty} R^i$ is a subset of trans(R). We will first

   show by induction that R^n is a subset of trans(R)
   for every "n" greater than zero.

   a. (Basis) From definition 6, part 2, it is
      immediately apparent that R is a subset of
      the trans(R).

   b. (Induction) Assume R^n is a subset of
      trans(R), and n is greater than or equal to
      1. Let <s,t> be the member of R^{n+1}. Since
   $R^{n+1}=R^n R$, there exists some u such that u is the
   member of set B, <s,u> ϵ R and <u,t> is the
   member of R^n. By the induction hypothesis and
   the basis step, <s,u> is the member of trans(R)
   and <u,t> is the member of trans(R). Since
   trans(R) is guaranteed to be transitive it
   follows that <s,t> is the member of trans(R),
   thus establishing that R^{n+1} is a subset of
   trans(R). Since R is a subset of trans(R) for
   all n>=1, we conclude that:

   $$\bigcup_{i=1}^{\infty} R^i \subset \text{trans}(R)$$

2. trans(R) is a subset of the $\bigcup_{i=1}^{\infty} R^i$. We will first

   show that:

   $$\bigcup_{i=1}^{\infty} R^i$$

13

is transitive. Let $<s,t>$ and $<t,u>$ be arbitrary members of $\bigcup_{i=1}^{\infty} R^i$, then for some integers $W>=1$ and $Y>=1$, $<s,t>$ is the member of R^W and $<t,u>$ is the member of R^Y. Then $<s,u>$ is the member of $R^W R^Y$, and because we know that $R^W R^Y = R^{W+Y}$, $<s,u>$ is the member of $\bigcup_{i=1}^{\infty} R^i$ and therefore $\bigcup_{i=1}^{\infty} R^i$ is transitive. Because trans(R) is contained in every transitive relation which contains R, it follows that trans(R) is a subset of $\bigcup_{i=1}^{\infty} R^i$.

From part 1 and 2 we can write:

trans(R) $= R \bigcup R^2 \bigcup R^3 \bigcup R^4 \ldots\ldots$

by using the basic set properties.

Theorem 3: Let R be a binary relation on a set B which is of cardinality n. Then

$$\text{trans}(R) = \bigcup_{i=1}^{n} R^i$$

Proof: We will show that R^j is a subset of $\bigcup_{i=1}^{n} R^i$ for all $j>0$. Assume $<s,t>$ is the member of R^j, then there is a directed path of length j from s to t in the digraph $<B,R>$, and by deleting cycles from this path we can obtain a simple directed path going from s to t. Because, in a graph with n nodes, the longest simple path is limited to length n, it

14

follows that <s,t,> is the member of R^i for some n>=i>0.

Thus R^j is the subset of $\bigcup_{i=1}^{n} R^i$ for j>0.

Definition 8: If R is a binary relation on a set B, then san:R denotes trans(R), the transitive closure of R, and fan:R denotes transitive reflexive closure of R (trans-refl(R)).

Now we will go into the theorems that will be helpful in finding the asymptotical time complexity behaviour of the algorithms.

Definition 9: Let f and g be functions and let their domains be the set of natural numbers and the codomains be the set of real numbers, then g asymptotically dominates f, or f is asymptotically dominated by g, if there exists s>=0 and t>=0 such that $|f(n)|=<t|g(n)|$ for all n>=s.

Example: Let f(n)=n/2 and g(n)=n^3 for all natural numbers "n", the above definition is satisfied by setting the s=0 and t=1, hence g asymptotically dominates f.

Definition 10: The set of all functions which are asymptotically dominated by a given function h is denoted by O(h), and is read as "big-oh of h", or "order h". If a given function say j is the element of O(h), then j is said to be O(h).

Theorem 4: Let the functions f, g, r be the kind of functions which map the natural numbers to the real numbers. Then:

1. f is O(f).

2. If f is O(g) then c*f is O(g) for any real number c.

3. If f and h are both O(g), then their sum (f+h), (where (f+h)(n)=f(n)+h(n).) is O(g).

Proof:

1. To show f asmptotically dominates f, we choose s=0 and t=1 and apply definition 9. Thus by definition 9 f dominates f.

2. If f is asymptotically dominated by g, then for some natural number m, k and for n greater than or equal to k, absolute value of f(n) will be less than or equal to the product of the m and g(n), i.e.:

$$m*|g(n)| >= |f(n)|$$

If we multiply both sides of this inequality by an arbitrary real number c the inequality remains the same. Now let m*c be equal to real number z then we rewrite the inequality as below:

$$z*|g(n)| >= c*|f(n)|$$

where z >=c. So by definition 9, c*f(n) is O(g).

3. Suppose f and r are both O(g), then there exists natural numbers, q, a, z, x such that $q*|g(n)| >= |f(n)|$ in the case n is greater than or equal to a, $z*|g(n)| >= |r(n)|$ in the case n is greater than or equal to x. Now assume Q=q+z and G=a (where a>x). Then we write:

$$q*|g(n)| + z*|g(n)| >= |r(n)| + |f(n)| = |f(n) + r(n)|$$

or

$$Q*|g(n)| >= (r+f)(n)$$

So r+f is O(g).

We usually represent the time consumed by an algorithm by a complexity function, say g, then O(g) is called the asymptotical time complexity behaviour of the algorithm. Note that the functions that have the same asymptotical time complexity behaviour may not cost us the same. Suppose the complexity function of an algorithm is the integer multiple of the time complexity function of another algorithm and suppose both algorithms have the same asymptotical time complexity behaviour. Clearly the first algorithm is more expensive than the other but they have the same asymptotical time complexity behaviour. So while using the asymptotical time complexity behaviour as the measure we have to be careful. In order to make this fact clear we will give an example.

17

Example:

Suppose two algorithms D and E have the complexity functions g and h respectively and let these complexity functions be,

$$g = K*m + C1$$

$$h = L*(m^2) + C2$$

where K = 40*L (constants)

C1 = C2 (constants)

Then for m =< 40 the algorithm E is less costly than the algorithm D and for m>40 the algorithm D is less costly than the algorithm E but the complexity function of algorithm D asymptotically dominates the complexity function of the algorithm E. So if we choose the algorithm D by only looking at its asymptotical time complexity behaviour and if the "m" does not take on values greater than 40, we would lose time instead of saving time.

From this point on we will use the order notation in which the explicity specification of the function is written in the parenthesis rather than the name of the function, so that $O(n^2)$ denotes the set of functions that are asymptotically dominated by $f(n) = (n^2)$.

We can write the classes of different complexity in order of increasing complexity as follows:

$$O(c) \subset O(\log n) \subset O(n) \subset O(n(\log n)) \subset O(n^2) \subset O(c^n) \subset O(n!)$$

Definition 11: If poly(m) is a polynomial of degree s, then poly(m) is $O(m^s)$.

2. <u>The Extensional Representations of Relations</u>

There are several representation techniques for representing relations. Among them, incidence matrix (or adjacency matrix), adjacency list [Ref. 5] and table representations [Ref.2] are the most common ones. There exists other representation techniques which are inherently the same as the techniques given above.

We can define the incidence matrix of a relation as follows: Let R be a relation with m tuples where m>=1. The incidence matrix of relation R is a 2-dimensional mXm array, say M, with the property that M[j,k]=1 if and only if the tuple <Aj,Ak> is in relation R, where the individual Ai belongs to the codomain and the individual Aj belongs to the domain of the relation R.

From an incidence matrix one can readily determine if a tuple is in the relation in question. In general the algorithms that work on the incidence matrix representation of relations have $O(n^2)$ time complexity behaviour, and if the incidence matrix of a relation is sparse the space utilization is not efficient. We will discuss this issue in detail later in the storage complexity analysis of the extensional representation techniques. A sample incidence matrix is shown in Figure 1.

a	R
π	3.1416
R	C
ʇʀ'	2
l	m
a	C
π	m

Relation W

	R	3.1416	C	2	m
a	1	0	1	0	0
π	0	1	0	0	1
R	0	0	1	0	0
ʇʀ'	0	0	0	1	0
l	0	0	0	0	1

Figure 1.   The Incidence Matrix of Relation W

20

In our system we will use a representation technique which is very similar to the incidence matrix representation, namely Hash-Incidence-Vector representation. We can define the incidence vector as follows: Let R be a relation with p tuples where p>=1, the incidence vector of relation R is a bit vector, say B, with the property that, if there are n distinct individuals in the domain of relation R and if the i'th individual of the domain and the k'th individual of the codomain are in relation with each other under the relation R, the $(k-1)*n+i=J$'th bit of B is 1 otherwise it is 0. Let the cardinality of the domain be n and the cardinality of the codomain be s, then the length of B is equal to (n*s) or in terms of number of memory locations it is equal to (ceiling((n*s)/C)) where C is the memory word length. In the hash incidence vector representation the domain (RIM set) and codomain (LEM set) individuals are represented by records in the separate hash tables, i.e., the LEM set individuals are represented by the records in a hash table called Left Hash Table and the RIM set individuals are represented in a hash table so called Right Hash Table (RHT). The records of the LEM set individuals are further linked to each other establishing a linked list structure in the LHT and similarly for the RIM set individuals. Each record mentioned above has

21

a field in which an integer to be used in index computation is stored. The Hash-Incidence-Vector for the relation W is shown in Figure 2.

In the adjacency list representation the m rows of the incidence matrix are represented as m linked lists. There is one list for each domain individual. The nodes in list i represent the individuals that are in relation with the individual i of the domain set. Each node has at least two fields, one of these fields represents the individual that is in relation with the i'th individual of domain and second field being a link field is used to construct the linked list structure. A sample configuration of an adjacency list is shown in Figure 3.

The table representation of a relation is the simplest representation technique with respect to others. We can define this representation technique as follows: Let R be a relation with m tuples, the table representation of R is a 2Xm array, say M, with the property that M[k,1]=Ai and M[k,2]=Aj where Ai and Aj are the individuals of the k'th tuple (<Ai,Aj>) of the relation R and Ai belongs to the codomain and Aj belongs to the domain of the relation R.[*]

_____

[*]Because our relational operations are defined by using the notations used by Russel and Whitehead in Principia Mathematica to #56, the order of domain and codomain will (unussually) be reversed.

22
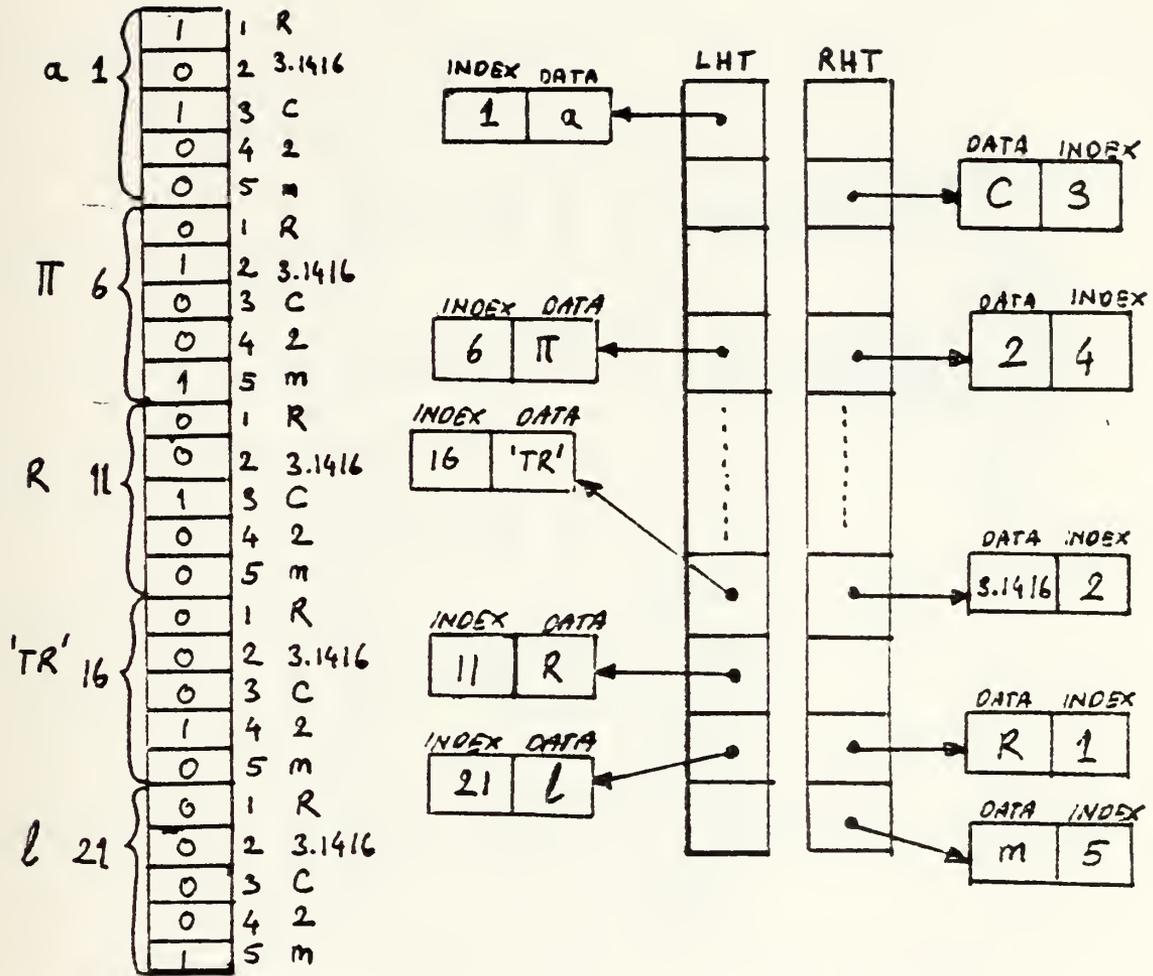
Figure 2.   The   Hash-Incidence-Vector   Representation   for
            Relation W

23
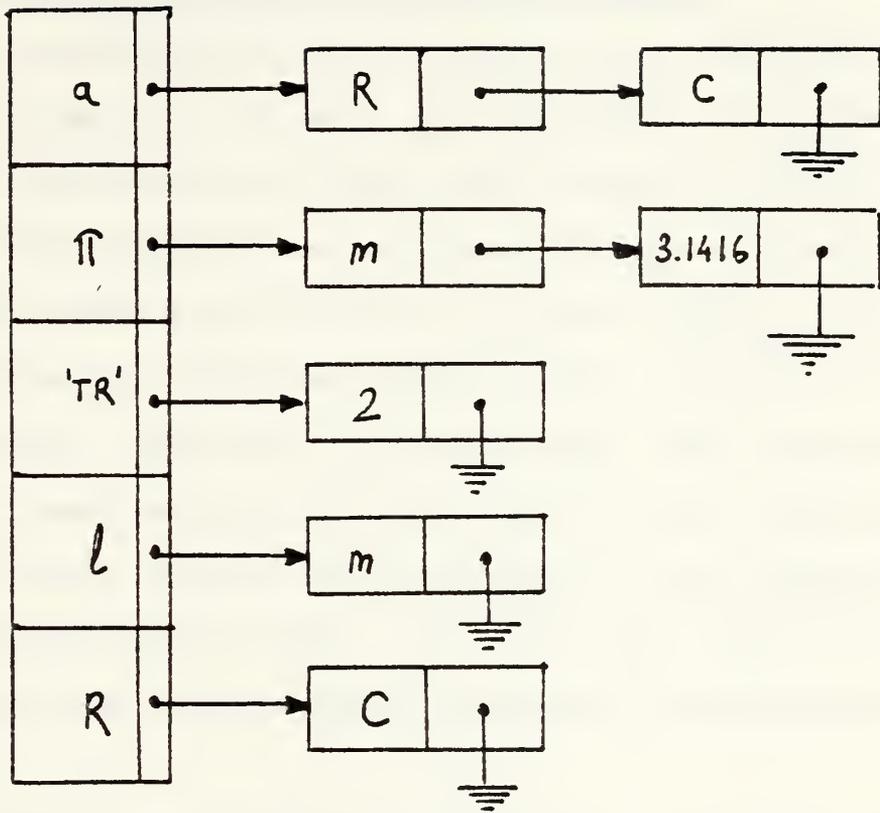
Figure 3.   The Adjacency List Representation for Relation W

Sometimes it is beneficial to represent the table of a relation as a linked list of records rather than an array. Each record of the linked list represents a tuple of the relation in question. A relation represented in this kind of representation is shown in Figure 4.

3. The Extensional Representations of Sets

One representation of sets is to represent the members of a set in a binary tree structure in which the members are represented as nodes. An alternative to the tree representation is based on the hash coding, in which the members of a set are stored in a hash table. In this representation the storage usage is poor because in some instances we may have a lot of unused hash table entries. In addition we need additional links that thread the records and/or hash table entires corresponding to set members, in order to do set operations. On the other hand in this representation the membership test operation becomes constant time.

Another representation technique is a bit vector representation of sets. In this representation technique a linearly ordered set, the so called Universal set (from which all sets are created) is represented in an array or linked
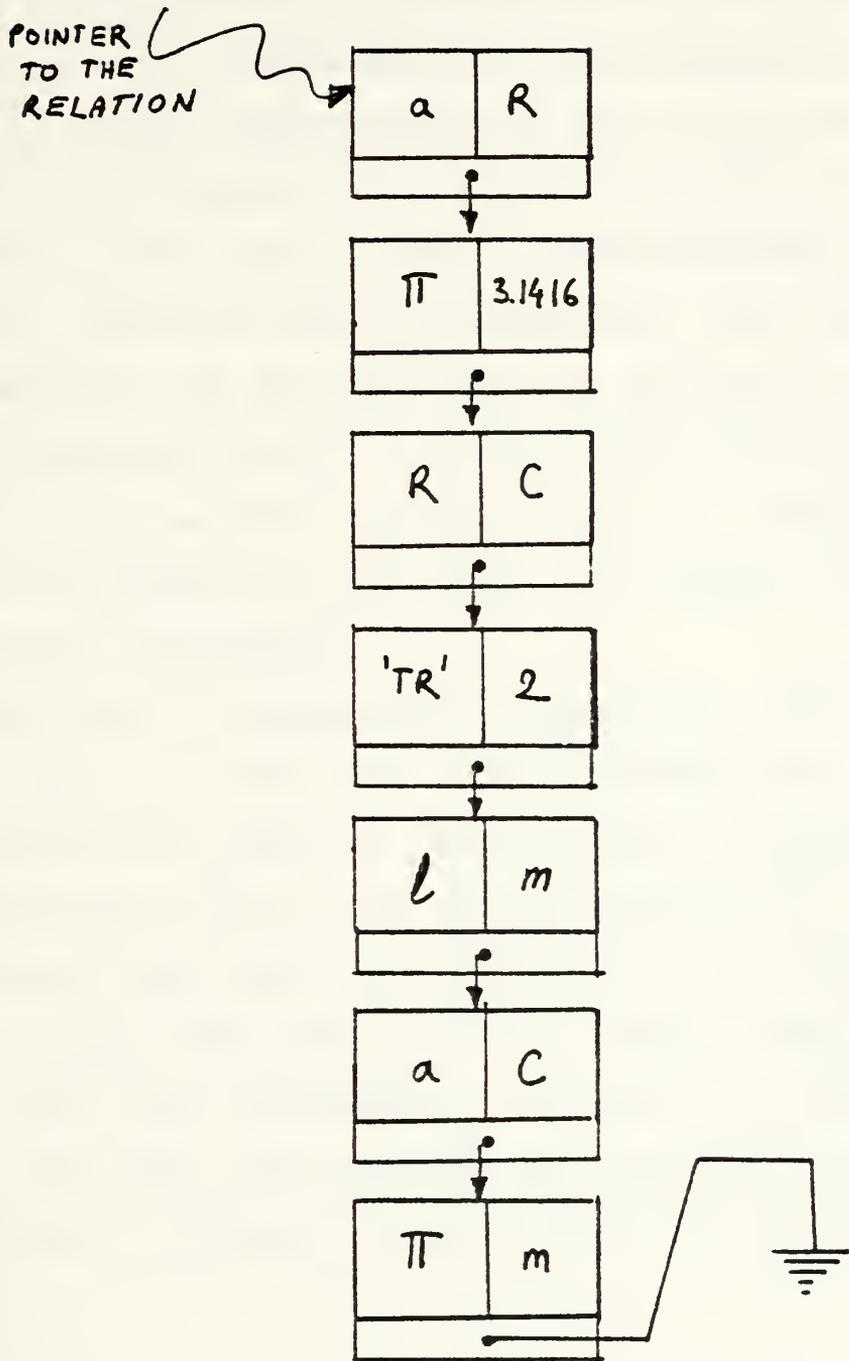
Figure 4. The Table Representation for Relation W

26

list structure and all the other sets that are subsets of this set are represented by bit vectors. So a subset C of the universal set UNIV is represented as a bit vector of m list where m is the cardinality of the universal set. Let's say the bit vector representing the set C is A; the K'th bit of the A is 1 if and only if the K'th individual of UNIV is the member of the set C. This representation has many advantages. First of all, the membership test becomes very easy. Secondly, the set operations can be done by using fast logical operations (and, or, union, not) on bit vectors. Further in the case the bit vectors are not largely sparse, the space utilization is efficient with respect to the other representation techniques.

The last representation technique, and the most common one, is the list representation. In this representation the amount of memory needed is proportional to the cardinality of the set being represented, and there exists linear time algorithms for doing set operations. In a practical sense some algorithms are slightly expensive even though their time complexity behaviour is linear. For example, the union and intersection operations require time proportional to the sum of the cardinalities of the operand sets.

27

Most of the time, it is beneficial to use a dynamic data structure rather than a static one in this representation technique. The linked list representation of a set is shown in Figure 5.

B. THE STRUCTURE OF THE SYSTEM

In this chapter we will describe the extensional representation system that we propose and will discuss the various properties of the system.

1. Consideration in Selecting a Set Representation

We have described the representation techniques for sets before; we will use the features of the list representation and the hash representation rather than one of the other representations. The prime reason for doing this is, our relational operations produce a significant number of intermediate sets and we have to represent the sets so that we use as large portion of the memory as needed. So we can not use the hash representation by associating a hash table with each set created. But also we would not want to lose the constant time membership test opportunity, so we will use a system wide hash table in which each individual of a set is represented as a record which is connected to the hash table entry which this individual hashes into. Furthermore, the
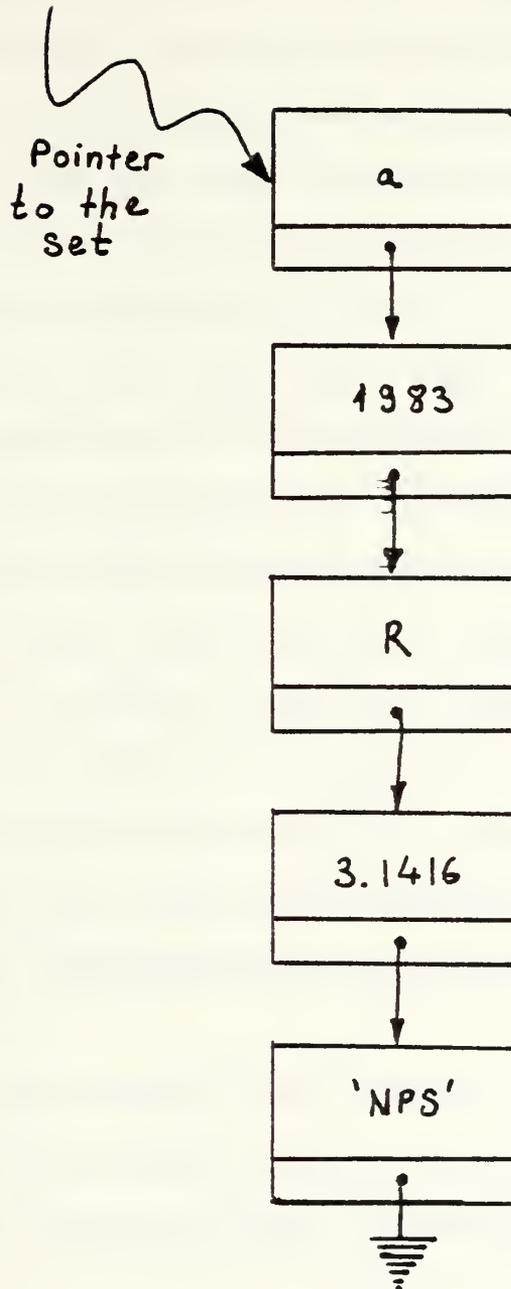
Figure 5. The Linked List Representation of a Set

individuals that belong to the same set are linked to each other in a linked list structure. This hash table will be explained further, later in this section. So the resulting representation technique has features from both the hash representation and the list representation.

We need to maintain the sets as linked lists in this hash table in order to be able to keep track of the individuals in a set and many relational operations are required to examine all the members of a set. Why don't we select the tree representation? Because the membership test and insertion is more costly than the hash representation. In addition, since the relational operations will produce a lot of sets, it would be costly to execute an O(log(n)) time algorithm to insert in the right place in the tree constructed so far the record for each individual of a set being produced by a relational operation.

2. Space Considerations for Selecting Relation Representation

In this section we intend to discuss the space requirements of various relation representation techniques. We will mainly focus on three representation techniques:

a.   Hash-Incidence-Vector representation.

b.   Table representation.

c.   Adjacency List representation.

These representation techniques have been explained in the previous section.

According to the current definition of the operations in our relational language there will be many references to the various relations perhaps even in a one line of the program, so if we consider the density of these references we would not want the underlying memory management system to access to the disk most of the time. This implies that the storage requirements of a particular representation technique become very critical. So even though our main intent is to analyze the time requirements implied by the representation techniques on the algorithms of the relational operations, we do not want to select those representation techniques that are in the first place infeasible in the space considerations.

a.   Storage Requirements of the Incidence Vector Representation:

As we explained before, there exist (m*n) entries of the incidence vector for representing a relation which has

31

a LEM set with the cardinality "m", and a RIM set with the cardinality "n". However, we packed this incidence vector into $(m*n)/C$ memory locations where "C" is the memory word length. The conditions under which maximum storage wasting occurs is the first question we should ask ourselves. In the worst case the cardinality of the LEM set and the cardinality of the RIM set become equal, let's say "n". In this case there should be at least "n" 1's in the incidence vector and $(n^2)-n$ 0's, so the overhead is:

$$(n^2) - n \text{ bits.}$$

We said there should be at least "n" 1's in the incidence vector, because every LEM set individual is in relation with at least one RIM set individual and analogously every RIM set individual is in relation with at least one LEM set individual. If this weren't the case, we would ask the question, How did that individual come to be inserted into the RIM (or LEM) set of the relation if it is not in relation with any individual in the LEM (or RIM) set. This can not occur, since, in the creation of the relation we put those individuals in the LEM set of the relation which are in relation with at least one RIM set individual, and in the same manner we put those individuals in the RIM set of the

relation that are in relation with at least one LEM set
individual and we create the incidence vector according to
the cardinalities of the LEM set and the RIM set of the
relation. So in general the incidence vector of a relation
can not contain less than "k" 1's where:

$$k = \max(m,n)$$

m = The cardinality of the LEM set of the
relation.

n = The cardinality of the RIM set of the
relation.

So in general we compute the overhead in the worst case by
using the formula below:

$$\text{number of unused bits} = m*n - \max(m,n)$$

In fact we can not consider the relation represented by the
incidence vector alone, since we are actually representing
the LEM and the RIM set of the relation along with the
incidence vector in our Hash-Incidence-Vector representation.
Hence the number of fields of each record representing a LEM
set or a RIM set individual should be taken into account. In
addition, the relation has a record in a hash table, which we
will call the relation table[*], and we have to add the space
occupied by that record to our cumulative formula. We write

---

[*]We will explain the structure of this table in
subsection 3.

33

the space complexity formula for the Hash-Incidence-Vector representation in terms of number of bits required, as follows:

$$fl = ceiling((m*n)/C) + K*(m + n) + D$$

where:

m = The cardinality of the LEM set of the relation.

n = The cardinality of the RIM set of the relation.

K = The number of bits required by each RIM/LEM set record in the RHT/LHT.

D = The number of bits required by the record of the relation in the relation table.

b. The Storage Requirements for the Adjacency List Representation

This representation technique is very dynamic and uses a large portion of the memory as needed. We will investigate what would happen in the worst case. In the worst case the relation may be a universal relation on its LEM and RIM set, which means that each RIM set individual is in relation with all the LEM set individuals of the relation. Therefore, we need (m*n) records to represent this kind of

34

relation. As we stated in the Hash-Incidence-Vector representation case, we can not assume that the relation is represented by the Adjacency List alone, so in order to make a fair comparison between the Hash-Incidence-Vector representation and Adjacency List representation we will assume that the RIM set records of the relation are represented in the RHT as it was in the Hash-Incidence-Vector representation case and that the records of the left individuals that are in relation with one or more right individuals are connected to the RHT records of these right individuals in the linked list structures. In fact, in this representation some relational operations are very costly; for example in order to obtain the LEM set of the relation being represented in this manner we must trace through all the linked lists of the kind explained above. So under the time considerations we would want to represent the converse of that structure in the LHT also, which makes some of the algorithms simpler than they otherwise would be. But let's assume we only assume the space requirements and we did not do that. In the worst case of the Adjacency List representation the cumulative storage requirement for representing a relation is given in terms of the number of bits below:

$$f2 = K*n + L*(m*n) + D$$

where:

> m = The cardinality of the LEM set of the
>   relation.
>
> n = The cardinality of the RIM set of the
>   relation.
>
> K = The number of bits required by each RIM
>   record.
>
> L = The number of bits required by each linked
>   list record (Our general set structure
>   record).
>
> D = The number of bits required by the record of
>   the relation in the relation table.

   c. Storage Requirements for the Table Representation

The Table Representation requires more storage than the Adjacency List representation most of the time, including the worst case when the relation is a universal relation on its LEM set and the RIM set. This is because the adjacency list representation removes the duplicates of the individuals in the right column of the table. In the table representation each tuple of the relation is represented as it is, and that causes the duplication of the right individuals and the left individuals in the columns of the table (if we look at the linked list structure of the table as conventional table). The Adjacency list representation does not represent a right individual in more than one place.

We write the cumulative storage requirements of the Table Representation in the worst case and in terms of number of bits as follows:

$$f3 = T*n*m + D$$

where:

n = The cardinality of the RIM set of the relation.

m = The cardinality of the LEM set of the relation.

T = The number of bits required for each table record.

D = The number of bits required for the record of the relation in the relation table.

d.   Comparison of Storage Requirements

Now we have to compare the formulas we found for the various representation techniques.   In fact the Hash-Incidence-Vector representation always requires the same amount of storage for a given "n" and "m", so actually we are comparing the worst case requirements of the other representation techniques with the fixed requirement of the Hash-Incidence-Vector representation.   Let's subtract "f1" from "f2", we find:

$$f2-f1 = L*m*n - K*m - (m*n)/C$$

If we factor out the "m":

$$f2-f1 = m*(L*n - K*m - n/C)$$

37

and equate the left side to 0, we obtain:

$$L*n - K - n/C = 0$$

and we find:

$$n = K/(L - (1/C))$$

This means that:

$$n > ceiling(K/(L-(1/C)))$$

and in the worst case of the Adjacency List representation, Hash-Incidence-Vector representation is always better than the Adjacency List representation. If we assume that the memory word length and the pointers are 16 bits then according to our definition of the fields of the records,

$$K = 80$$
$$D = 32$$
$$C = 16$$
$$T = 48$$

then n should be greater than 3. Because we have indicated that the Table Representation requires more storage than the Adjacency List representation, there is no need to do the comparison for the table representation. So we conclude that in the worst case the Table representation and the Adjacency List representation dominate the Hash-Incidence-Vector representation in storage consideration. In addition to that, as one of the "m" or "n" becomes smaller than the other the number of redundant bits in the incidence vector decreases. Suppose n<m so there are m 1's in the incidence

38

vector but there are (m*n) entries and (m*n)<(m*m). On the other hand we can not expect the worst case to occur every time under the practical considerations, so, if the relation being represented is a bijective function (one to one, and onto), clearly the redundancy in the incidence vector becomes maximum. This is the best case for the Table Representation and the Adjacency list representation. Let's rewrite the functions f1, f2 and f3 under this case:

(note that in this case n=m)

f1 = 2*K*n + (n*n)/C + D

f2 = (K + L)*n + D

f3 = T*n + D

if we subtract f1 from f2 and equate the result to ^, and if we solve "n" in the resulting equation, we find that:

n = C * (L - K)

Hence as long as L<K the result is negative that shows us in that case the Adjacency List representation is better than the Hash-Incidence-Vector representation. If we subtract f1 from f3 and do the same steps we find:

n = C * (T - 2*K)

This means the table representation is much better than the Hash-Incidence-Vector representation in this case. Lastly, if we do this for f2 and f3, we can not find n. In that case

we apply the numerical values (in our case) to the constants
for comparison; if we substitute the values given for L, K
and T and compare f1 and f2 as lines having different slopes
we see that:

$$f1 = 112*n + D$$

$$f2 = 48*n + D$$

Therefore, in this case the Table representation is better in
storage than both the Hash-Incidence-Vector representation
and the Adjacency List representation.

Selection of one of these representation
techniques under these space considerations depends greatly
on our expectations on the kind of relations that we will be
working on. For example if we are working on bijective
functions the most appropriate representation technique is
the table representation. If we are representing trees by
using binary relations, the most appropriate technique is the
Adjacency List representation. However, because of the
nature of our system the Hash-Incidence-Vector representation
becomes attractive. We will explain this by giving an
example.

Example:

Suppose we have a relation which has the LEM set
cardinality 100 and the RIM set cardinality 200. By
substituting these values for the constants in the formulas

for fl and f2,. we find the storage requirements for the Adjacency List representation (in the worst case of Adjacency List representation) to be 82 Kbyte and the storage requirements for the Hash-Incidence-Vector representation to be 5.5 Kbyte. In the best case of Adjacency List representation (a bijective function), the storage requirements for the Hash-Incidence-Vector representation remains the same, but the storage requirements for the Adjacency List representation drops to 2.4 Kbyte.

If we think about the above example, in the best case of the Adjacency List representation we do not gain much, but in the worst case we lose a lot.

In the analysis of the algorithms we will mainly focus on the Hash-Incidence-Vector representation, and the Table representation. We will inspect the relative efficiency of using Hash-Incidence-Vector representation instead of using Table representation in terms of time. We will not look into the Adjacency list representation, because the Hash Incidence Vector representation is essentially the same as the Adjacency List representation in which the linked lists are represented as bit strings. This analogy is demonstrated in Figure 6.
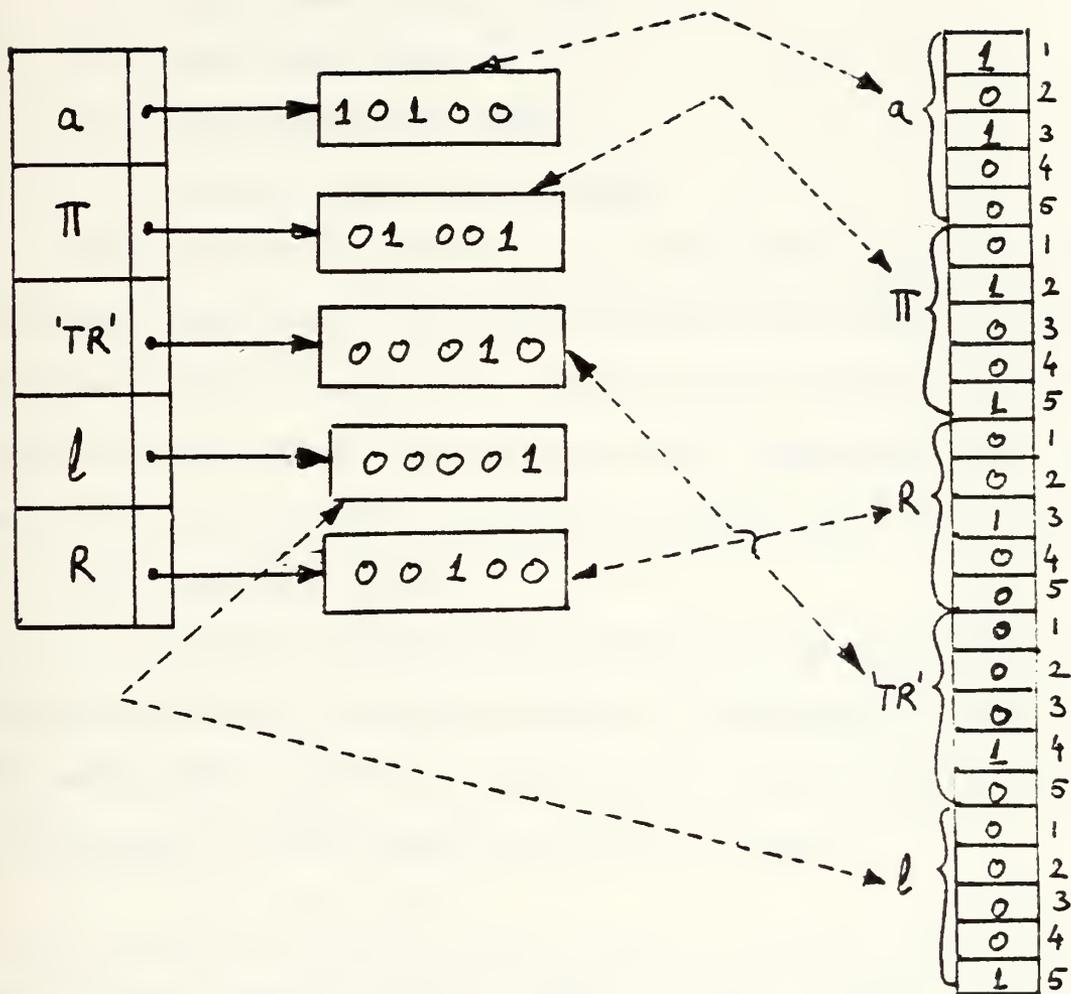
Figure 6. The Analogy Between the Adjacency List Representation and the Hash-Incidence-Vector Representation

42

3. System Hash Tables and Their Structure

The system consists of six hash tables which are:

a. Relation Table (RT).

b. Left Hash Table (LHT).

c. Right Hash Table (RHT).

d. Set Table (ST).

e. Set Hash Table (SHT).

f. Scratch Hash Table (SCHT).

The system handles the collisions by using the bucketing technique. In this technique the records of the individuals that hash into the same hash table entry are linked to each other in a linked list structure and this linked list is connected to the hash table entry in question.

a. Relation Table

In this table each relation known by the system is represented by a record which is connected to the hash table entry into which the identifier of the relation hashes. The structure of the record is shown in Figure 7.

Rid	PFLM	PFRM	PCOLS	BASE	|RIM|	|LEM|	COLLINK

Figure 7. The Relation Table Record Structure

The Rid field of the record contains the character string representing the relation's identifier. The PFLM field

contains a pointer which points at the first record of the Left Members set (LEM) of the relation, which resides in the Left Hash Table. The PFRM field contains a pointer to the first record of the Right Members Set (RIM) of the relation, which is in the Right Hash Table (RHT). PCOLS is also a pointer field, which contains a pointer to the code (function) representing the relation; its use will be explained later in Section III. The BASE field contains the beginning address of the buffer allocated for the incidence vector of this relation. The |RIM| field of the record contains an integer which is the cardinality of the Right Members Set (RIM) of the relation. The |LEM| field contains an integer which is the cardinality of the Left Members Set (LEM) of the relation being represented. The COLLINK field contains a pointer to the record of the relation which has been hashed into the same hash table entry as a result of collision. The structure of the Relation Table (at one point in execution) is shown in Figure 8.

  b. Left Hash Table

    This hash table contains the records of the LEM set individuals of the relations. The LEM set records of a relation are linked to each other in a linked list structure. The structure of a LHT record is shown in Figure 9.

44
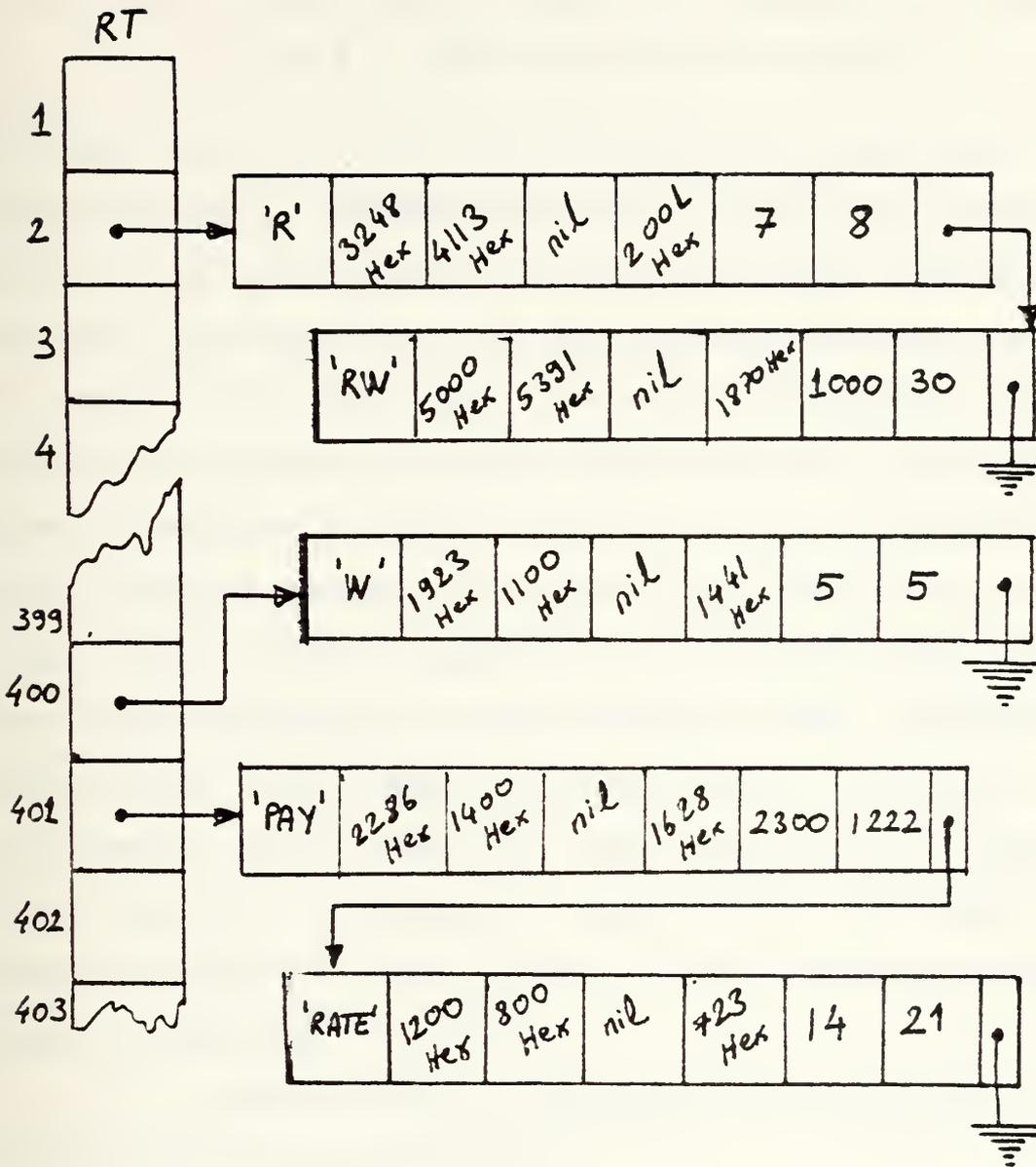
Figure 8. The Appearance of the Relation Table at One Point in Execution

45

Rid	TASE	PML	INDEX	COLLINK	PRRM

Figure 9.   The LHT Record Structure

The Rid field of the record contains a pointer to the character string representing the relation's identifier.   The inclusion of this field is necessary in order to distinguish the same individuals of the LEM sets of different relations. The TASE field contains a pointer to the next LEM set individual's record.   The PML field contains a pointer to the memory location where the individual being represented by this record is stored.   The INDEX field contains an integer which will be used in computing the indices of the Incidence Vector corresponding to the individual being represented by this record.   The COLLINK field contains a pointer to the individual's record which has been hashed into the same hash table entry as a result of collision.   The PRRM field contains a pointer which points at the related right member's record in the RHT.

The LHT has an associated hash function that we will call "Left Hash Function".

c. Right Hash Table

The "Right Hash Table" has exactly the same structure as the LHT. The only difference is, it contains the records of the Right Members Set individuals of the relations. It has an associated hash function that we will call "Right Hash Function". Figure 10 shows the arrangement of the LHT, RHT in combination with RT at one point in execution.

d. Set Table

This hash table contains the records of the set which are known by the system. The record structure of this table is shown in Figure 11.

The Sid field of this record structure contains the character string representing the set identifier. The CARD field contains an integer which is the cardinality of the set being represented. If this field contains -1, then the cardinality of the set has not been computed. The COLLINK filed contains a pointer to the record of another set (if any) which has been hashed into the same hash table entry as a result of collision. The PSS field contains a pointer which points at the first individual's record of the set in question. This record is the beginning record of the linked list structure that represents the set. As we mentioned earlier the records of the individuals are also connected to the entries of the Set Hash Table which will be explained

47
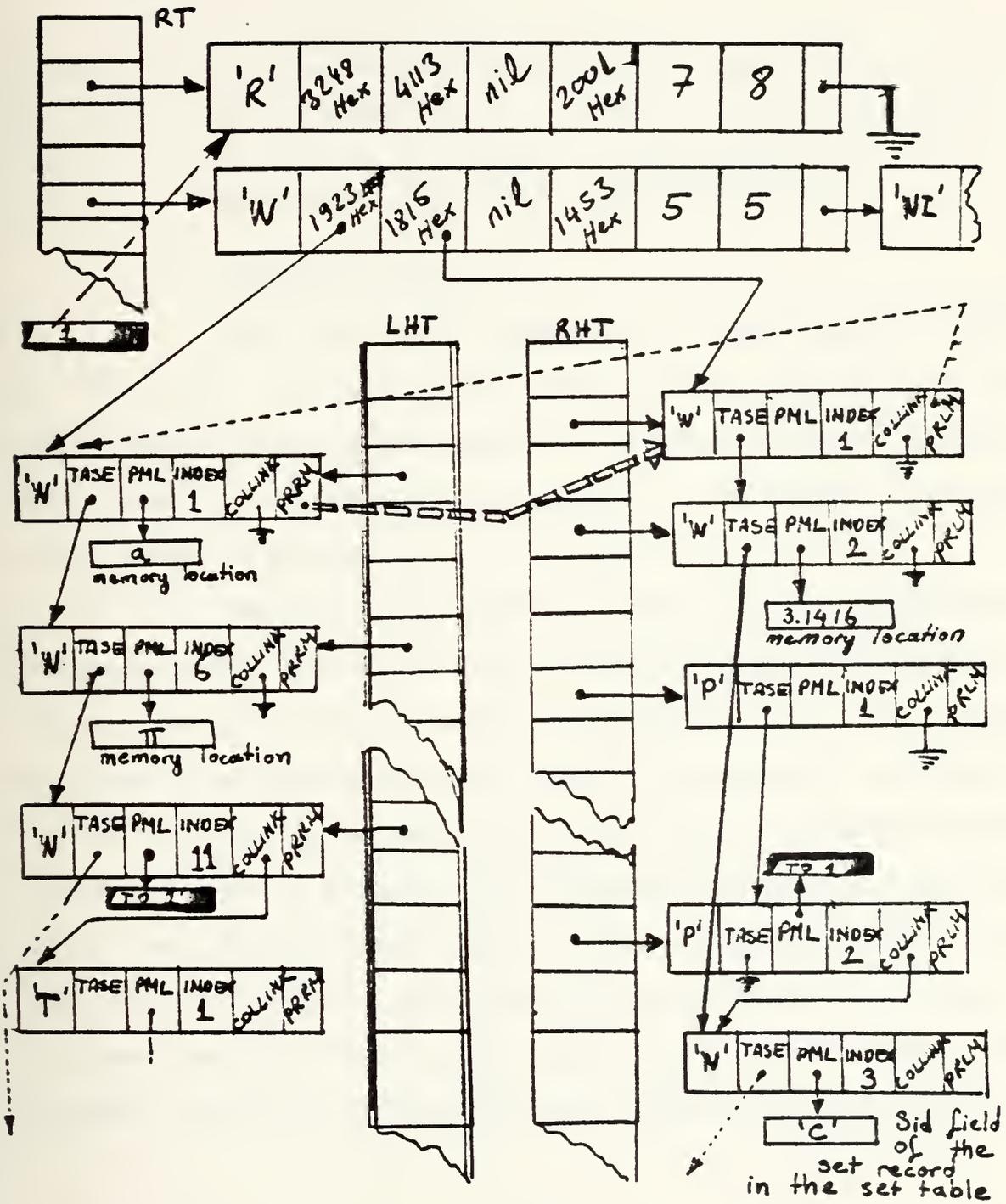
Figure 10. The Arrangement of RT, LHT, RHT at One Point in Execution

48

next. The Set Table has an associated hash function, which we will call "Set Hash Function". Figure 12 shows the arrangement of the Set Table at one point in execution.

Sid	CARD	PSS	COLLINK

Figure 11. Set Table Record Structure

e. Set Hash Table

This hash table contains the individuals' records of the sets. As explained before the records of the individuals that are the member of the same set are linked to each other in a linked list structure. The record structure is as shown in Figure 13.

The Sid field of this record structure contains a pointer to the character string which represents the set to which the individual being represented by this record belongs. Inclusion of this field is necessary in order to distinguish the same individuals of different sets. The PML field contains a pointer to the memory location where the individual being represented by this record is stored. The TASE field is another link field which contains a pointer to the next record of the linked list structure of the set. The COLLINK filed, as it was before, contains a pointer which

49
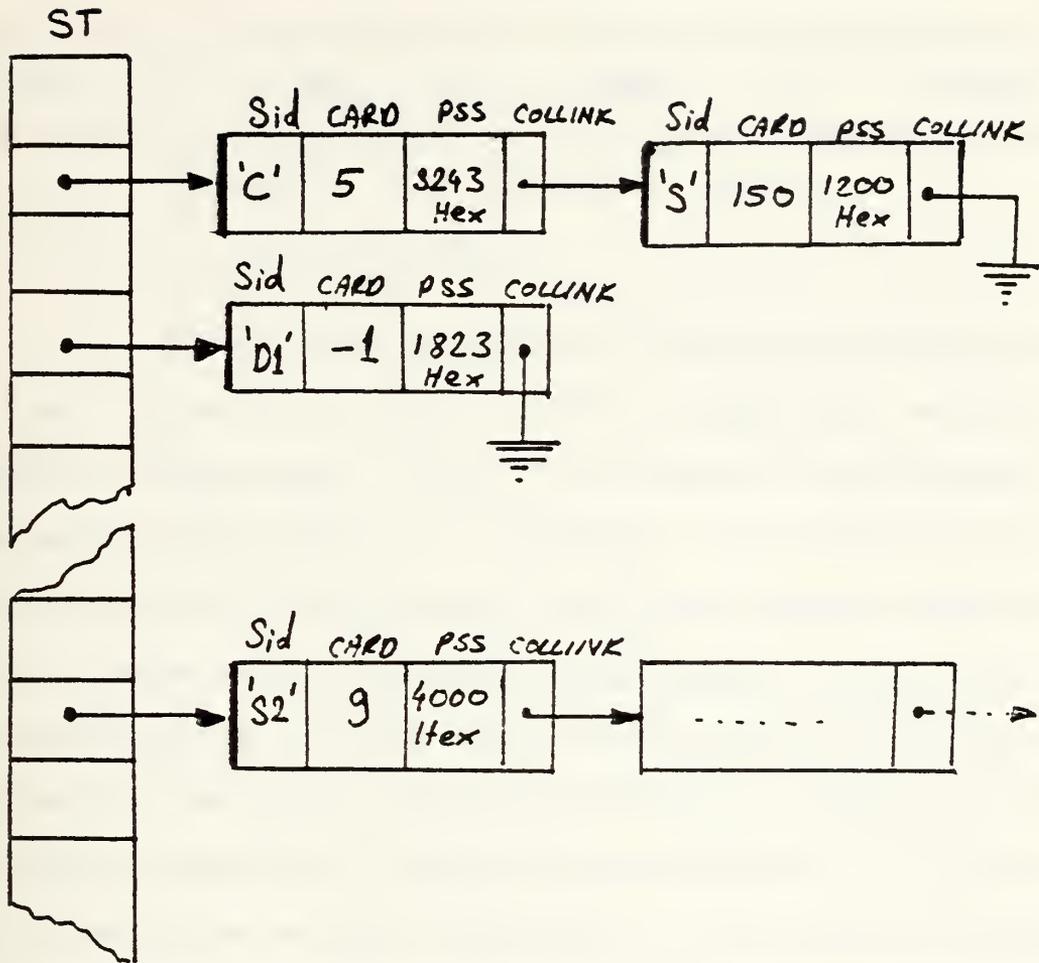
Figure 12. The Appearance of the Set Table at One Point in Execution

goes into the collision chain. Figure 14 shows the arrangement of the Set Hash Table at one point in execution.

Sid	PML	TASE	COLLINK

Figure 13. The SHT Record Structure

f. Scratch Hash Table

The Scratch Hash Table is exactly the same as the Set Hash Table and is used to store temporary sets during relational operations. After the operation terminates, the records of this table are disposed. We could use the Set Hash Table for this purpose, but doing pointer updates in such a crowded table becomes very complex. In addition the SCHT need not be as large as the other hash tables since it is used for only one operation and is cleaned up for a subsequent operation. So the record density in this table will be very low and the possibility of collisions decreases. It is the implementer's decision to continue to use the Set Hash Table for this purpose or not. Note that if the SHT is used for this purpose, the creation of temporary set identifiers (for the temporary sets) becomes necessary.

4. Hash Functions

As we mentioned before, each system hash table has an associated hash function. We will assume that the reader is already familiar with Hash Coding and collision handling

51
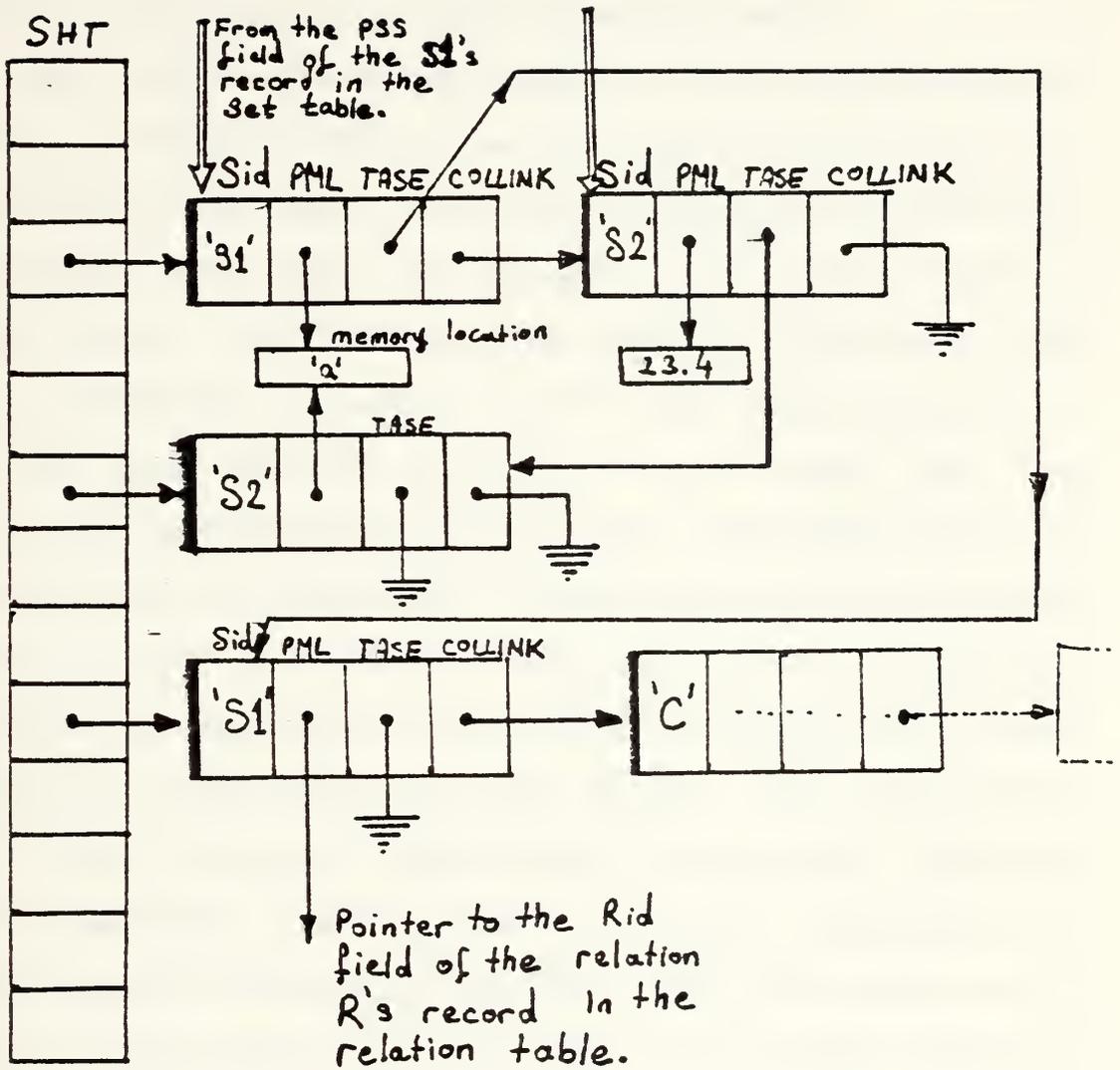
Figure 14.   The Arrangement of the SHT at One Point in Execution

techniques. In addition, we will not go into the detail of the hash functions; i.e., we will not consider how the index of a hash table entry is computed corresponding to an identifier since this is an implementation issue.

The hash function associated with the Relation Table (RT) takes a relation identifier and maps it into the index of a Relation Table entry. The hash function associated with the Set Table (ST) takes a set identifier and maps it into a Set Table entry. The hash function associated with the SCHT takes an individual and maps it into a SCHT entry and so on.

The hash functions associated with the LHT, RHT, and SHT have slightly different properties. The hash function associated with the LHT takes the individual and the relation identifier (which identifies the relation that the individual in question belongs to) and concatenates them; then it maps the resulting identifier to a LHT entry. This is done in order to have a better distribution in the LHT. The hash function associated with the RHT has the same properties as the hash function associated with RHT, only the table we are hashing is the RHT instead of LHT. The hash function associated with the SHT takes the individual and the identifier of the set which this individual belongs to and concatenates them; then it maps the resulting identifier to an SHT entry.

Each hash function mentioned above tries to find the record of the individual in the collision bucket. (If any collision occurred before, there will be a collision bucket connected to the hash table entry to which the hash function mapped us.) If it finds the record of the individual, it returns the pointer to the record of this individual; otherwise, it returns the pointer to the last record of the collision bucket or the hash table entry found (if there is no individual connected to this hash table entry). Of course, it will inform the caller about the kind of pointer returned.

In our system the identifier of the individual is the individual itself; hashing functions view the individuals as bit strings and compute the indices of the hash table entries by using these bit strings. Another important property of our system is the individuals of a relation or a set may be of different types. For example a set may contain a relation or another set as a member. So in our system the relations, sets, integers, characters, character strings, bit strings, reals, etc. are all individuals. This type independency is achieved by maintaining the pointers to the memory locations where the individuals are actually stored, in the data fields of the hash tables' records, rather than the individuals themselves.

5. Referencing the Incidence Vector

Before we explain the way we reference the incidence
vector of a relation, we will explain how we arrange the
integers stored in the index fields of the LEM set and RIM
set records of a given relation. Given a relation, the
integers associated with the LEM set individuals begin with 1
and increase by K where K is the cardinality of the RIM set
of the relation; i.e., if the cardinality of the RIM set of
the relation is 3 and the cardinality of the LEM set of the
relation is 2, the integer stored in the index field of the
first LEM set individual's record will be 1, and the integer
stored in the index field of the second LEM set individual's
record will be 4, and so on. The integers stored in the
index fields of the RIM set records begin with 1 and increase
by 1; i.e., in the above example the integer stored in the
index field of the first RIM set individual's record will be
1, the integer stored in the index field of the second RIM
set individual's record will be 2, and so on. The beginning
address of the incidence vector of the relation is stored in
the BASE field of this relation's record in the Relation
Table (RT).

Now we have to explain how we reference the incidence
vector of a relation. Suppose we are given a tuple and a
relation The question is whether this tuple is in the given
relation or not. We first hash with the right component

55

individual to the RHT and find its record, then we hash with the left component individual of the tuple to the LHT and find its record. We extract the integers stored in the INDEX fields of these records and add them up; then we subtract 1 from the result and obtain the INDEX of the incidence vector entry corresponding to this tuple. Let's call the resulting INDEX, "K". (Of course, if we can not find records for the individuals above, the question can be answered immediately.) In the next step we extract the beginning address of the incidence vector from the record of the relation. Let's call this address BASE. Then we call the algorithm below with the BASE and K being the arguments. Algorithm reference (K, BASE):

1. Offset = ceiling(K/C).
2. Location = offset + BASE - 1.
3. Fetch the contents of the memory location by using the address computed in step 2.
4. h = K - (offset*C) + 1.
5. Extract the h'th bit from right and test it. If it is 1 return true, else return false.

In the above algorithm C is the memory word length, "offset" is a variable of type integer, "location" is a pointer variable, "h" is a variable of type integer. We needed to do the above computations because we pack the n bits of the

56

incidence vector into ceiling(n/C) memory locations, where C
is the memory word length.

6. Table Representation

Another representation technique that we will be
focusing on is the table representation. We will represent
the table of a relation as a linked list of records in which
each record represents a tuple of the relation. The record
structure is as shown in Figure 15.

LEFT	RIGHT
LINK	
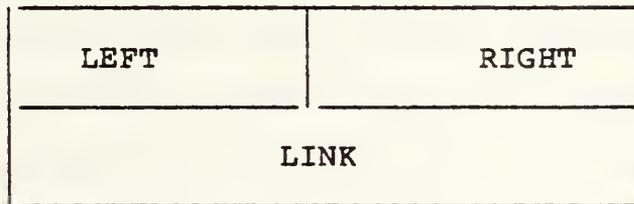
Figure 15. The Structure of the Table Records

The LEFT field contains a pointer to the memory location
where the left component of the tuple (which is an
individual) is stored. The RIGHT field contains a pointer to
the memory location where the right component of the tuple is
stored. The left component of the tuple belongs to the Left
Members Set of the relation (or in other words Codomain of
the relation) and the right component of the tuple belongs to
the Right Members Set of the relation (or in other words the
Domain of the relation). The LINK field contains a pointer
to the record which represents the next tuple of the
relation.

57

We will not define a complete environment for the table representation. Of course the environment defined for the Hash-Incidence-Vector representation (i.e., The RT, ST, SHT) could be used in this case too.

7. <u>About the Algorithms</u>

We will write our algorithms in English step by step. In the time complexity analysis we will refer to the steps of the algorithms and associate the terms of the complexity functions with the steps. The comments will be written in parenthesis between the steps of the algorithms. Sometimes we will insert loops as steps into the algorithms, which are written in a PASCAL-like algorithmic language. This is done to make the algorithm clear to the reader.

In the time complexity functions we will use the capital letters to represent the constants and the small letters to represent the variables. Even though we will be inspecting the worst case asymptotical time complexity behaviour of the algorithms, and constants do not affect the asymptotical time complexity behaviour of the algorithms, we will provide the complexity functions of the extensional algorithms with the predicted explicit constants in Appendix B. In predicting these constants we will make some assumptions. For example hashing to a hash table requires 10 memory references to be made. Even though we will not define an explicit environment for the Table representation we will

58

assume we use the same environment that we will use for the Hash-Incidence-Vector representation in predicting those constants. This is necessary in order to do a fair comparison between the Table representation and Hash-Incidence-Vector representation.

II.   ANALYSIS OF EXTENSIONAL ALGORITHMS

In this section we will define some of the relational operations' algorithms that work on the extensional representation structures and we will determine the worst case asymptotical time complexity behaviour of these algorithms.

The reader can find the analysis of the remaining relational operations' algorithms in Appendix A.

A.   FUNCTION APPLICATION (F:x)

Given an individual we want to apply a function to that individual in order to find the corresponding individual in the codomain of the function. We know that functions are in fact left univalant relations.   This means given an individual in the domain there exists a unique individual corresponding to that individual in the codomain or that no individual exists in the codomain corresponding to the individual in the domain. Now we have to state that fact more carefully.

Definition: Let A and B be sets.   A function f from A to B, denoted:

f:A->B

is a relation from A to B such that for every aϵA, there exists at most one bϵ_ such that <a,b>ϵf.   In this case we write:

```
f:a = b
```

On the other hand in the relations case, given an individual in the domain of the relation we may find more than one individual that is in relation with that individual in the codomain of the given relation.

In our case the domain is the RIM set of the relation/function and the codomain is the LEM set of the relation/function. In our system function application operation is also defined for relations. This may seem dangerous to the reader, but we have other operations such as "Unit image" that returns the set of individuals in the codomain which are in relation with the given individual in the domain so it is the user's responsibility to use the apropriate operation when he/she is programming. The reason for doing this is we will treat function application in a special manner to make this operation faster (constant time) because the "Function Application" operation is a very frequently used primitive function of the system. If we check to detect if more than one individual exists in the codomain for the given individual in the domain, this operation becomes an order "n" operation in the hash incidence vector representation. So there is no need to accept an O(n) algorithm for this operation when there are other operations that serve the user in the relations case. For example if the user wants to learn the salary of an

employee by applying a relation (that relates the salaries to the employees) to the given employee name, it is obvious that the relation is a left univalant relation and he can do that without fear. In fact there exists many relations that are obviously left univalant and the user should be able to perform this fast operation on those relations. In addition to that, the user may want to use this operation instead of "Unit Image" operation even though it is known that the relation to be applied is not left univalant. If the user is not sure that the relation in question is left univalant or not, he should use the "Unit Image" operation to obtain the set of individuals that are in relation with the given individual, then he/she should apply the "Unit Class Selector" operation to the resulting set. This operation calls the "Error Handler" if the argument set is not a singleton set.

As we mentioned earlier, because we use this operation very frequently we have to reduce the time complexity of its algorithm to constant time. We do this by adding a pointer field to the RHT record structure, namely PRLM (Pointer to the related left member). The pointer in that field is set to the LHT record of the individual which is in relation with the individual being represented by the RHT record in question. In the same manner we allocate a pointer field in the RHT structure which we will call PRRM, that serves the

same purpose. Note that we still have to construct the incidence vector of a left univalant relation even though it may seem unnecessary at first glance; the reason is the converse of a left univalent relation is not necessarily a left univalent relation (except in the case of bijection) and in this case we have to treat this condition in the converse operation as a special case. In fact this is not the only reason; many operations that we will define algorithms for such as the "Relative Product" and "First Ancestral" operations, expect the argument relations to have incidence vectors. So rather than adding this case as a special case to each algorithm and constructing the incidence vector of a left univalent relation when it becomes necessary, we had better construct it the first time the relation is created. In fact both solutions have tradeoffs. If we construct the incidence vector of a left univalent relation the first time the relation is created and if we do not need that incidence vector in any operation in the program, we waste space. On the other hand if we maintain the code to construct the incidence vector of a relation in each operation's program which requires that the operand relations have their incidence vectors together with them, we waste space again. Which of the solutions is advantageous is an implementation decision.

63

The algorithm for the Hash-Incidence-Vector representation is as follows:

1. Get the argument individual.

2. Hash with that individual into the RHT under the given relation identifier (for the relation being applied to the given individual).

3. Find the RIM record of that individual in the RIM set of the relation in question.

4. Follow the pointer found in the PRLM field of that record and reach the record of the left individual in relation with the right individual in question.

5. Follow the PML field of the record found and extract the individual from the memory location where it is saved and return it.

The worst case (also the average case) asymptotical time complexity behaviour of this algorithm is obviously constant time (O(c)). Because no matter how large the relation is, we always make the same number of memory references.

Now we have to define the algorithm for the table representation. The algorithm is as follows:

1. Get the individual.

2. Start from the beginning of the relation's table; proceed down in the table record by record by following the links between the records.

3. For each record found, compare the argument individual with the individual represented by the "right" field of that record.

4. The first time a match is found, return the individual represented by the "left" field of the current table record.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The worst case complexity function of this algorithm can be written as:

$f = K*p + C$

where:

p = Table size/Relation size.

K = The constant number of memory references made for each table record found.

C = The constant number of memory references made by the housekeeping operations. (In this case C is very small because there is no need to update any global table.)

In the worst case the relation may be a universal relation on its LEM and RIM sets, or in other words the relation may be equal to the cartesian product of its LEM and RIM set. By

assuming the LEM and the RIM sets have the common cardinality "n", we can substitute:

    n*n

in place of "p" in the above function. So the complexity function becomes:

    $f = K*(n^2) + C$

So by looking at the exponent of the term with the larger exponent, we conclude that the above polynomial has the asymptotical behaviour of order 2, and in turn we conclude that the algorithm has the worst case asymptotical time complexity behaviour of $O(n^2)$.

B.   UNIT IMAGE ((unimg:R):x)

This operation, given an individual in the domain (RIM set) of the relation, returns the set of individuals that are in relation with the given individual in the codomain (LEM set) of the relation.

The algorithm for the Hash-Incidence-Vector representation is as follows:

1.   Get the argument individual.

2.   Get the relation identifier, hash with that relation identifier into the relation table and find the record of the relation in the relation table. Follow the pointer found in the PFLM field of that record, and find the first left member's record in the LEM set of the relation.

3. Hash with the argument individual into the RHT under the given relation identifier, find the RHT record of the argument individual and extract the contents of the "index" field of that record. Record the resulting integer in the temporary variable "temp1".

4. Start from the beginning of the LEM set of the relation (the first record is found in step 2), and proceed down in the LEM set record by record by following the TASE links between the records. For each record found in this manner extract the contents of the index field, reference the incidence vector of the relation with this index and the index stored in the variable "temp1" by using the "reference algorithm". If a 1 is found in the corresponding incidence vector location then, hash into the SHT with the current left individual under the set identifier which will be described in step 5, and establish a set record. Copy the PML field of the current LEM set record into the PML field of that record. If this is the first set record created, mark it with pointer "P". Link the set records created in this manner to each other by their TASE links. Keep a count beginning with 0 and increment it for each set record created, with 0.

5. Hash to the set table (ST) with the identifier of the resulting set, which is:

"uimg"(relation's identifier)(individual's identifier)

67

Establish the record of this set, put the pointer "P" into
the "PSS" field and put the last value of the count into the
"CARD" field of that record.

We write the worst case complexity function of that
algorithm as follows:

   f = K*n + C

where:

   n = The cardinality of the LEM set of the given relation.

   K = Constant number of memory references made for each
       LHT record found while proceeding in the LEM set of
       the relation in Step 4.

   C = Constant number of memory references made in steps 1,
       2, 3 and 5.

By looking at the exponent of the term with the larger
exponent we conclude that the worst case asymptotical time
complexity behaviour of this algorithm is O(n), where n is
the cardinality of the LEM set of the relation.

Now we have to define the algorithm for table
representation. The algorithm is as follows:

   1. Get the argument individual.

   2. Start from the beginning of the relation's table,
proceed down in the table record by record, by following the
links between the records. For each record found in this
manner, compare the argument individual with the individual
represented by the "right" field of that record. If a match

68

is found, hash into the SHT with the current left individual under the set identifier described in step 5 of the previous algorithm, establish a new set record and copy the "left" field of the current table record to the PML field of this set record. If this is the first set record created in this manner, mark it with a pointer and link the set records created in this manner to each other by their TASE links. Keep a count beginning with 0 and increment it for each set record created.

3. Continue to do step 2 until the end of the table of the relation.

4. Update the set table as it was done in step 5 of the previous algorithm.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We write the worst case asymptotical time complexity of that algorithm as:

$f = K*p + C$

where:

P = Relation size.

K = Constant number of memory references made for each table record found in step 2 of the algorithm.

C = Constant number of memory references made by the housekeeping operations (such as the number of memory references made in step 4).

69

We know that in the worst case:

p = n*n

where:

n = The common cardinality of the LEM set and the RIM set
of the relation in question (Assumption).

So we rewrite the complexity function as:

$f = K*(n^2) + C$

So by looking at the exponent of the term with the larger exponent we conclude that the worst case asymptotical time complexity behaviour of this algorithm is $O(n^2)$.

C. CONVERSE OF A RELATION (Rc)

We formally express the converse of a relation as:

Rc = {<x,y>| <y,x> ε R}

So if D is the digraph of R, the digraph of Rc can be constructed from D by reversing the direction of all arcs of D. This can be done in the table representation of a relation by simply interchanging the columns of the table.

Since the converse of a relation is another relation it participates in the relational operations as the original relation does, so in the Hash-Incidence-Vector representation case we necessarily have to construct the incidence vector, the LEM set, and the RIM set of that resulting relation. The algorithm for the Hash-Incidence-Vector representation is as follows:

70

1. Get the identifier of the original relation.

2. Hash to the relation table, find the record of the relation, follow the pointers in the PFLM and in the PFRM fields of that record and find the records of the first left member and the first right member of the relation respectively.

3. Extract the contents of the |LEM| and the |RIM| field of the relations record, allocate a memory block as large as:

$$(|LEM| * |RIM|)/C$$

(Where |LEM| is the cardinality of the LEM set of the original relation and |RIM| is the cardinality of the RIM set of the original relation and C is the memory word length.)

4. Make a separate copy of the LEM set of the original relation in the RHT under the relation identifier, "Rc" (i.e., the records will contain identifier "Rc" in their "Rid" fields, where R is the identifier of the relation in question). Keep a RIM set index count and increment it for each record copied; put the updated value of that count into the index field of the record created each time a record is created.

5. Make a separate copy of the RIM set of the original relation in the LHT under the relation identifier "Rc". Keep a LEM set index count and increment it by the cardinality of the RIM set of the original relation. For each record

71

copied, put the updated value of the LEM set index count into the index field of each new record created.

6. Extract the integer found in the |RIM| field of the original relation's record in the relation table and call it K. Start from the beginning of the original relation's LEM set and proceed down in this set record by record. For each record found extract the integer stored in the INDEX field of this record and call it L. Call the original relation's incidence vector A and the new relation's incidence vector B and execute the loop below.

For j = L to L+K by 1 do:

$$B[j+|RIM|] = A[j]$$

end-do

7. Hash to the relation table with the new relation identifier "Rc", and establish the record for the new relation. Copy the |LEM| field of the original relation's record into the |RIM| field of that record; in the same manner copy the |RIM| field of the original relation's record into the |LEM| field of that record. Put the beginning address of the new incidence vector into the BASE field of that record, put the pointers to the records of the first left member and first right member of the resulting relation into the PFLM and the PFRM fields of that record respectively.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We write the worst case asymptotical time complexity function as:

$f = P*(m*n) + L*n + M*m + C$

where:

m = The cardinality of the LEM set of the original relation.

n = The cardinality of the RIM set of the original relation.

L = The constant number of memory references made while copying each record of the RIM set.

M = The constant number of memory references made while copying each record of the LEM set.

P = The constant number of memory references made while copying the entries of the original incidence vector to the corresponding entries of the new incidence vector.

In the above function the first term corresponds to step 5, the second term corresponds to step 4, the third term corresponds to step 6, and the last term (the constant C) corresponds to the other steps of the algorithm.

Let:

m = n

S = L + M

73

then the complexity function becomes:

$$f = P*(n^2) + S*n + C$$

In step 6 of the algorithm we had to make a number of memory references proportional to the square of n, where n is assumed to be the common cardinality of the LEM and the RIM sets of the original relation, as a result of copying the (n^2) bits of the original incidence vector to the (n^2) bits of the new incidence vector. So, by looking at the degree of the term with the largest exponent we conclude that the worst case asymptotical time complexity behaviour of this algorithm is $O(n^2)$.

Now we have to consider how this operation could be performed on the table representation. Obviously the algorithm is simpler in this case. The algorithm is as follows:

1. Start from the beginning of the table of the relation, and proceed down in the table. For each table record found create a new table record. Copy the "left" field of the original record into the "right" field of the new table record. In the same manner, copy the "right" field of the original record into the "left" field of the new record. Link the new table records created in this manner to each other by their "link" fields.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

74

In the worst case the original relation may be a universal relation on its LEM and RIM sets. Assuming the LEM set and the RIM set cardinalities are equal to "n" the relation size becomes equal to the square of "n", so we write the worst case complexity function as:

$$f = K*(n^2) + C$$

where:

    n = The common cardinality of the LEM and the RIM set of the original relation.

    K = The constant number of memory references made for each table record of the original relation in step 1.

    C = The number of memory references made by the housekeeping operations such as updating the relation table, etc.

So we conclude that the worst case asymptotical time complexity behaviour of that algorithm is $O(n^2)$.

D. SET OPERATIONS

    1. Set Union (R∪S)

        The union of two sets contains those members which are in either one of the two operand sets or both. We can formally express that as:

$$R \cup S = \{ x \mid x \varepsilon R \text{ OR } x \varepsilon S \}$$

        Taking the union of two sets may involve a lot of comparisons and exhaustive searches in the inorderly structured linked lists of the operand sets. Concatenating

the linked lists of the two sets and then removing the duplicates may be one solution but removing duplicates is a very expensive operation. In addition we have to preserve the original sets while we are obtaining the union of them, so we have to make separate copies of the operand set structures and perform the operations on those copies.

Our solution for this problem is to use the properties of the SHT. By establishing the resulting set in the SHT the duplicates are automatically removed.

The algorithm is as follows:

1.   Hash with the first and second operand set identifiers to the set table, find their records, follow the PSS fields of those records, and find the beginning records ⌐f the two operand sets.

2.   Start from the beginning of the first set's linked list and proceed down in the linked list record by record.   For each record found, hash into the SHT with the individual represented by that record under the new set identifier "RUS".   Establish the record of that individual in the SHT only if there is no record for that individual in the SHT already.   Link the records of the individuals in the SHT by their TASE links as they are created.

3.   Start from the beginning of the other set's linked list and proceed down in the linked list record by record.   For each record, hash with the individual

represented by that record into the SHT under the new set
identifier. Establish its record in the SHT if there is no
record for that individual in the SHT already.

4. Hash to the set table with the set identifier
"RVS" where R is the identifier of the first operand set and
S is the identifier of the second operand set. Establish the
record of that set in the set table and put the pointer to
the linked list structure established in the SHT into the PSS
field of this record.

Note that the order of the operand sets is arbitrary
so if we establish the record of the set under the identifier
"RVS", a subsequent reference to the set "SVR" may cause the
same set to be reconstructed again. Of course we do not want
that, so we have to accept a convention and let the system
realize that convention. We assume that when a reference to
the union of two sets is made, the system first looks up the
set table for the record of this set, if it is not already
present there, it takes the identifier of the referenced set
apart, with the character "V" being the pivot character, and
interchanges the operand set identifiers with the character
"V" being in the center. Then the system hashes with the
resulting identifier to the set table and looks for the
record of that set. If there is no record for that set in
the set table, it executes the algorithm given above.

77

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

Clearly the algorithm goes through both operand sets once. Assuming the cardinalities of the operand sets are equal, the worst case time complexity function of that algorithm can be written as:

$$f = 2*K1*n + K2*m + C$$

where:

  n = The common cardinality of the operand sets.

  m = The cardinality of the second argument set = n.

  K1= The number of memory references made for each record found in the set while proceeding in the set in step 2.

  K2= The number of memory references made for each record found in the set while proceeding in the set in step 3.

  C = The number of memory references made in steps 1 and 4.

Let:

$$K = K2 + K1$$

then the worst case complexity function becomes:

$$f = K*n + C$$

So clearly, the worst case asymptotical time complexity behaviour of this algorithm is $O(n)$.

2. Set Intersection (R/\S)

The intersection of two sets contains those members which occur in both operand sets. That can be formally written as:

$$R/\backslash S = \{x | x \epsilon R \land x \epsilon S \}$$

We will use the SHT mechanism for this operation, like we did in the "set union" operation. The algorithm is as follows:

1. Hash with the identifiers of the operand sets to the set table, find their records, follow the pointers in the PSS fields of those records and find the first records of the linked list structures of those sets.

2. Start from the beginning of the first operand set (order is not important) and proceed down in the linked list of the set record by record. For each record found hash into the SHT with the individual being represented by that record under the second operand set's identifier. If this individual also has a record in that set structure, hash to the SHT with this individual again, but this time under the new set's identifier "R/\S" and establish the record for that individual in the SHT, if there is no record for that individual in the SHT already. Link the records of the individuals created in the SHT to each other by their TASE links as they are created. Keep a cardinality count beginning with 0 and for each record created increment this

count. Mark the first record of the resulting set structure
with the pointer P.

3. Hash to the set table with the identifier of the
resulting set, which is "R/\S", establish the record of that
set in the set table, put pointer P into the PSS field of
that record, and put the last value of the cardinality count
into the "CARD" field of that record.

Now we will do the worst case asymptotical time
complexity analysis of this algorithm.

We write the worst case complexity function of that
algorithm as:

$$f = K1*n + C$$

where K1, C, n and m mean the same as the corresponding
parameters defined in the set union operation.

Let the cardinality of the operand sets be equal,
then the complexity function becomes:

$$f = K1*n + C$$

So we conclude that the algorithm has the worst case
asymptotical time complexity behaviour of O(n).

3. <u>Set Difference (R-S)</u>

The difference of two sets, R and S contains those
members which are in R but not in S. This can be written
formally as:

$$R - S = \{x | x \epsilon R \text{ and not}(x \epsilon S) \}$$

We will use SHT for this operation too. The algorithm is as follows:

1. Hash with the identifiers of the sets to the set table, find their records, follow the pointers found in the PSS fields of those records and find the first record of each set.

2. Start from the beginning of the first operand set's linked list (call it R). (* The first operand set will be accepted and called the reference set because the set difference operation is not commutative. *) Proceed down in the linked list record by record. For each record found, hash with the individual represented by that record into the SHT under the second operand set's identifier and check if a record of this individual exists in that set structure. If so do nothing, else hash with this individual to the SHT under the new set identifier. Create a record for this individual in the SHT. Link the records created in this manner to each other by their TASE links as they are created.

3. Hash to the set table with the new set identifier "R-S" and establish its record and put the pointer to the beginning of new linked list structure into the PSS field of that record. Put the last value of the cardinality-count into the "size" field of that record.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We write the complexity function as follows:

f = K1*n + C

where K1 and C are as defined in the union operation case. In this case n represents the cardinality of the first operand set; the cardinality of the second operand set does not have any affect on the complexity function because we make K memory references for each record of the first argument set in step 2. Of course this is true in the case the second operand set is already in the SHT. Since this operation takes only those sets that are known by the system as operands and we represent the sets that are known by the system in the SHT, the mechanism is well defined. Thus we conclude that the worst case asymptotical time complexity behaviour of that algorithm is O(n).

E. INITIAL MEMBERS (init:R)

The initial members of a relation are the left members that are not right members. This can be stated formally as:

init:R = { x | for some y, xRy and not yRx }

Our "init" operation is supposed to take a relation identifier as argument and return a set of individuals that are the initial members of the given relation.

The algorithm for "Hash-Incidence-Vector" representation is given below:

1. Get the relation identifier.

82

2. Hash with that relation identifier into the relation table; find the record of that relation.

3. Follow the pointer found in the PFLM field of that record and find the LHT record of the first left member of that relation.

4. Find the individual's identifier by following the pointer found in the PML field of that record.

5. Hash into the RHT with the individual's identifier found in step 4 under the relation identifier in question. If there is no RHT record for that individual in the RHT, then hash with that individual into the SHT under the new set identifier ("init:"(relation identifier)), and establish its record. If it is the first record established in this manner then mark it with a pointer.

6. Find the next individual's record by following the pointer found in th TASE link field of the current record in the LHT; repeat steps 4 and 5 for that individual.

7. Repeat steps 4, 5 and 6 until the LEM set of the relation is exhausted. As the records are created in the SHT link them to each other.

8. Establish the record of the set created above in the Set Table under the identifier "init:R", where "R" is the identifier of the relation in question. Put the pointer P (that was set in step 5) into the PSS field of that record.

Once we establish the record of this set in the set table any subsequent references made to this operation take constant time. Because we are trying to find the worst case behaviour we had to write the costly part of the algorithm.

As we can see, step 7 of the algorithm causes the worst case asymptotical time complexity behaviour of the algorithm to the $O(n)$, where "n" is the cardinality of the LEM of the relation.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We define the worst case time complexity function of that algorithm as:

$f = K*n + C$

where:

   n = The cardinality of the LEM set of the relation.

   K = The number of memory references made for each record of the LEM set of the argument relation in step 5.

   C = The number of memory references made in steps 1, 2, 3, 4 and 8.

By looking at the worst case time complexity function we conclude that the worst case asymptotical time complexity behaviour of this algorithm is $O(n)$.

Now we have to define the algorithm that works on the table representation. Suppose our algorithm begins to examine the individuals on the left column one by one and for

84

each one performs an exhaustive linear search in the right column to decide if the individual in question is also present in the right column. This would obviously be an order two, $O(p^2)$ algorithm, where p is the relation size. But we may use the SHT mechanism again to reduce the time complexity of the algorithm.

The steps of the algorithm are given below:

1. Start from the beginning of the left column. Proceed down in the left column of the table by following the link fields of the table records and by looking up the individual from the "left" field of each table record. In fact the individual is not directly obtainable from the "left" field because one level of indirection is involved. That means it has to follow the pointer found in the "left" field of that record in order to find the individual. For each individual hash into the SHT under the new set identifier described in step 5 of the previous algorithm, establish its record and link the records as they are created in SHT. If the record is the first record created in this manner mark that record with the pointer P.

2. After the left column is exhausted start from the beginning of the right column and proceed down in the right column. For each individual found in the right column, hash into the SHT with that individual under the new set's identifier. If the record of that individual is already

present in the SHT, delete it. After deletion update the links between the records created in the SHT appropriately.

  3. Establish the resulting set's record in the set table (as is done in the set operations' algorithms). (* We did not explain the steps of that algorithm in detail, because the steps are similar to the steps of the algorithm defined for the set operations. *)

This algorithm requires one exhaustive linear search of the left column and one exhaustive search of the right column.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The worst case complexity function of that algorithm can be written as:

$f = K*p + M*p + C$

let $N = K + M$, then the function becomes:

$f = N*p + C$

where:

  p = Relation size (table size or equivalently the number of tuples in the relation).

  n = The cardinality of the LEM set of the relation.

  N = The number of memory references made (averaged) in each iteration of step 1 and 2.

  C = The number of memory references made by the housekeeping operations.

We know that in the worst case the relation may be equal to the cartesian product of its LEM set and the RIM set, which means the relation size is equal to the product of the cardinalities of the LEM set and the RIM set of the relation. So in the worst case we see that:

p = n*m

where:

n = The cardinality of LEM set.

m = The cardinality of RIM set.

let n = m, then:

p = n*n

If we apply this result to the above complexity function, it is obvious that the algorithm has the worst case asymptotical time complexity behaviour of $O(n^2)$.

F.   RIGHT RESTRICTION (R\C)

It is often useful to limit the domain of a relation. This operation, given a set and a relation, restricts the RIM set of the given relation to the given set. We can express the effect of this operation as follows:

R\S = { <x,y>| <x,y>εR /\ yεS }

So we bind the domain of the relation R to the intersection of the domain and the set S. It is clear that the operation should extract those individuals which are not in the given set from the RIM set of the relation.

We will first define the algorithm for the Hash-
Incidence-Vector representation. The algorithm is as
follows:

1. Get the relation identifier and the set identifier.
Find their records in the relation table and in the set table
respectively by hashing to those tables with these
identifiers.

2. Follow the pointer found in the PFLM field of the
relation's record and find the record of the first left
member in the LHT. Begin from the beginning of the linked
list structure of the LEM set of the relation, and proceed
down in that linked list record by record. For each record
found, hash to the LHT with the individual represented by
that record, under the relation identifier:

(Relation identifier)'\'(set identifier)

and establish its LHT record. Link the records created in
this manner to each other as they are created in the LHT.
(* This step effectively makes a separate copy of the LEM set
of the original relation, which becomes the LEM set of the
new relation. *)

3. Test if the given set is represented extensionally or
intensionally by following the pointer in the PSS field of
the set record. If it is detected to be extensionally
represented do the steps below:

88

a. Follow the pointer found in the PFRM field of the relation's record. Start from the beginning of the relation's RIM set; proceed in this set record by record; for each record found, hash into the SHT with the individual being represented by this record under the argument set identifier. If a record for this individual is present in this set structure, make a separate copy of the RIM set record of that individual in the RHT (like was done for the left individuals' records in step 2) under the new relation's identifier. Link the records created in this manner in the RHT by their TASE links. (* This step effectively copies those RIM set records of the original relation which represent some individual in the argument set, into the new RIM set of the new relation. *)

b. Hash to the relation table under the new relation's identifier. Establish the record of this relation in the relation table with the new relation identifier being in the "Rid" field. Copy the |LEM| and the BASE fields of the original relation's record into the |LEM| and the BASE fields of the new relation's record respectively. (* So the new relation makes use of the original relation's incidence vector. *)

4. If the argument set is detected to be intensionally represented do the steps below:

a. Start from the beginning of the linked list structure of the original relation's RIM set, proceed down in the RIM set record by record. For each record found in this manner test if the individual being represented by that record is a member of the argument set. (* This membership test will be explained further in the explanation of the intensional representation structures. *) If this individual is in the argument set then hash with that individual into the RHT under the new relation identifier and copy all the fields (except the TASE and Rid fields) of the RIM set record (which belongs to the original relation) into the new record's corresponding fields. Put the new relation's identifier into the Rid field of that record. Link the records created in this manner to each other by their TASE links as they are created.

b. Do step 3-b.

Now we will do the worst case asymptoticaly time complexity analysis of this algorithm.

The right restricted relation makes use of the original relation's incidence vector, which significantly reduces the time complexity of the resulting algorithm. We expect that most of the time the cardinality of the argument set will be smaller than the cardinality of the RIM set of the relation, but of course that may not be true all the time, i.e., we do not have a restriction on the cardinality of the argument

90

set. In the worst case the argument set may be a super set
of the RIM set of the original relation, in that case we have
to copy all of the RIM set of the original relation in order
to obtain the resulting relation's RIM set. (So the
resulting relation becomes exactly equal to the original
relation.) Under these considerations we write the worst
case time complexity function of that algorithm as:

$f = K*m + L*n + C$

where:

  m = The cardinality of the LEM set of the original
      relation.

  n = The cardinality of the RIM set of the original
      relation.

  K = The constant number of memory references made while
      copying each LEM set record.

  L = The constant number of memory references made while
      copying each RIM set record.

  C = The constant number of memory references made by the
      housekeeping operations.

In the above function the first term corresponds to step 2,
the second term corresponds to step 3, and the last term
corresponds to the other steps of the algorithm.

91

Let m=n and Z=K+L then the complexity function becomes:

f = Z*n + C

Thus it is clear that the algorithm has the worst case asymptotical time complexity behaviour of O(n).

Now we have to define the algorithm for the table representation. The algorithm is as follows:

1. Start from the beginning of the linked list structure of the relation's table and proceed down in the table record by record by following the links beteen the records. For each record found in this manner hash with the individual represented by the "right" field of that table record into the SHT under the argument set identifier. If a record of that individual is already present in the SHT then create a new table record (that will belong to the restricted relation). Copy the "right" and "left" fields of the original relation's record to the corresponding fields of the new record. Link the new table's records created in the above manner to each other as they are created.

The algorithm seems simpler than the previous one, but in the worst case we can not say it is less costly.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

In the worst case the relation may tbe a universal relation on its LEM set and RIM set (note that this is different from saying "the relation is the universal relation

92

on its MEM set"). This means it contains all of the tuples
that can be constructed from the LEM and RIM sets, and each
tuple of that relation will have its left component from the
LEM set and it will get its right component from the RIM set.
This relation is in fact the cartesian product of the LEM set
and the RIM set. In this case the number of tuples in the
resulting relation will be equal to the product of the
cardinalities of the LEM and the RIM set. By assuming that
the LEM set and the RIM set of the original relation have the
common cardinality "n" we write:

$p = n*n$

where "p" is the size of the relation. Note that in step 1
we get through the whole structure of the argument set and in
step 2 we get through the linked list structure of the
relation's table. In addition in the worst case the argument
set may be a super set of the RIM set of the original
relation; in that case we necessarily copy the whole table of
the relation, which means the restricted relation and the
original relation become exactly equal to each other. So we
write the worst case time complexity function of that
algorithm as:

$f = T*p + C$

where:

p = Relation size.

        T = The constant number of memory references made for
            each record of the relation in step 1.

        C = The constant number of memory references made by the
            housekeeping operations.

In the above function the first term corresponds to step 1 of
the algorithm. If we substitute (n^2) for "p" in the above
function we get:

    $f = T*(n^2) + C$

So the algorithm has the worst case asymptotical time
complexity behaviour of $O(n^2)$, where "n" is the common
cardinality of the LEM set and the RIM set of the original
relation.

G. LEFT RESTRICTION (C/R)

    It is often useful to limit the codomain of a relation,
so this operation takes a set identifier and a relation
identifier and restricts the LEM set of that relation to the
given set. This means that after this operation is performed
there remains only those individuals in the LEM set of the
resulting relation which are in the argument set. We can
state this as follows:

    $S/R = \{ <y,x> | <y,x> \epsilon R \wedge y \epsilon S \}$

The algorithms for "Left Restriction" operation are
essentially the same as the algorithms for the "Right
Restriction" operation for both representation techniques.

94

The only difference is we bind the LEM set in the Hash-
Incidence-Vector representation or left column in the table
representation instead of the RIM set in the Hash-Incidence-
Vector representation or the right column in the table
representation. So there is no need to rewrite the
algorithms and repeat the complexity analysis.

H.   RELATIVE PRODUCT (RS)

This operation takes two relation identifiers and
produces another relation which is the relative product of
the given relations in the order they have been given.

This operation has an expensive algorithm because each
tuple of the resulting relation may originate from the
presence of many different tuples in the argument relations.

The algorithm for Hash-Incidence-Vector representation is
as follows:

Let the first argument relation be R and the second
argument relation be S:

1.   Find the records of the argument relations in the
relation table by hashing with their identifiers to the
Relation table (RT).

2.   Make separate copies of the LEM set of the relation
R and the RIM set of the relation S in the LHT and in the RHT
respectively under the relation identifier "RS". While doing
that keep a LEM set index count and for each LEM set record
created, increment this count by the cardinality of the RIM

95

set of relation S. Each time a LEM set record is created, put the updated value of this count into the index field of this LEM set record. Establish the record of the new relation in the relation table under the relation identifier "RS". Establish the pointers to the LEM set and the RIM set of that relation into the PFLM and PFRM fields of the record respectively. Copy the |LEM| field of the relation R's record into the |LEM| field of the new record; in the same manner, copy the |RIM| field of the relation S's record into the |RIM| field of the new record. Allocate a block of memory as large as:

$$(|LEM| * |RIM|)/C$$

where C is the memory word length. Put the beginning address of this block into the "BASE" field of the new relation's record i RT. Initialize the new incidence vector. Establish the contents of |RIM| field of the new relation's record in the variable "CARD".

3. Start from the beginning of the LEM set of the relation RS, proceed down in the LEM set record by record by following the TASE links between the records. For each record found in that manner, extract the contents of the index field, put it in variable "beginl" and perform these steps:

a. Find the RIM set record of the first right member of relation R and proceed down in the RIM set of the relation R, by following the TASE links between the records.

b. For each tuple found which is being represented by the record pair found in steps 3 and 3-a, check if the relation R has this tuple by hashing with the components (individuals) of the tuple to the LHT and RHT and by using the reference algorithm. If the relation R does not have this tuple then do nothing. Else, hash to the LHT with the individual found in step 3-a under the relation S. If there is no LEM set record present for that individual in the LEM set of relation S then again, do nothing. Otherwise, take the index stored in the index field of the LEM set record of relation S and put it in variable "begin2".

c. Take the:

begin2 to begin2 + CARD

bits of the incidence vector of relation S and OR them with the:

begin1 to begin1 + CARD

bits of the new incidence vector.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The worst case time complexity function of this algorithm can be written as:

$f = K*n + L*m + S*n*q*(m/D) + R*(n*m)/D + C$

The first term of this function corresponds to copying the LEM set of the relation R in the LHT, where "n" is the cardinality of the LEM set. The second term corresponds to copying the RIM set of the relation S in the RHT, where "m" is the cardinality of the RIM set. The third term corresponds to step 3, where "q" is the cardinality of the RIM set of relation R, and constant D is the memory word length. The term:

m/D

stands for the number of memory references made for each OR operation. The fourth term corresponds to the initialization of the new incidence vector and the last term (constant C) is the number of memory references made by the remaining steps of the algorithm, such as establishing the new relation's record in the relation table.

The cost of fourth term may be reduced by pipelining, and the cost of the third term can be reduced by putting a large portion of the incidence vector of relation S into the cache memory.

Let $n=m=q$, $W=(S/D)$, $Y=(R/D)$ and $T=K+L$, then the complexity function becomes:

$$f = W*(n^3) + Y*(n^2) + T*n + C$$

Obviously the algorithm has the worst case asymptotical complexity behaviour of $O(n^3)$.

Now we have to write the algorithm that works on the table representation. It is natural to expect a more expensive algorithm by the experience we have had until now, but our task is to find out how expensive it is relative to the above algorithm. The algorithm is as follows:

Let the first argument relation be R, and the other be S.

1. Start from the beginning of the table of relation R; for each tuple of relation R:

    a. Look up the right individual.

    b. Search the left column of the relation S for that individual.

    c. If a tuple of relation S is found to have that individual as the left individual, hash into the SCHT with the left individual of the current tuple of relation R (i.e., the left individual of the tuple from which we get its right individual in step 1-a). Establish its record in the SCHT; if there is already a record of some left individual connected to this hash table entry, look up the individual being represented by that record. If it is the same individual do step 1-d, else search all the following neighbouring occupied hash table entries for the record of that individual. If it can not be found, establish the record of that left individual and connect it directly to the first neighbouring unoccupied hash table entry, which follows the hash table entry that the hashing function first found. If the record of that

99

individual is found to be connected to one of the neighbouring occupied hash table entries, again do step 1-d.

d. In any of the above cases, either the record of the left individual was found to be already present or was established in step 1-c; establish the record of the right individual of the current tuple found in S. Connect it either to the left individual's record or to the end of the bucket (if a bucket is already connected to the left individual's record). Continue to search for the tuples in S which have the right individual mentioned in step 1b as their left individual, and repeat the steps 1c and 1d for these tuples.

2. Repeat step 1 until the relation R is exhausted. Set a pointer to each hash table entry occupied and put that pointer into a temporary array of type pointer. (* The result of the steps 1 and 2 is the "adjacency list" representation of the resulting relation in the SCHT, now the remaining steps are to convert that representation to our table representation. *)

3. Do step 3 of the algorithm given for the table representation in the "union" operation for relations.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The worst case complexity function of that algorithm is as follows:

100

$$f = K*(p*p) + L*(n*m) + C$$

The first term of this function corresponds to steps 1 and 2 of the algorithm, where "p" is assumed to be the size of both argument relations. The second term corresponds to the step 3 of the algorithm where "m" is the cardinality of the RIM set of the relation S and "n" is the relation R's cardinality. We know that in the worst case:

$$p = n*n$$

where "n" is the cardinality of both the LEM set and the RIM set of the relation in question. Let m=n, then the complexity function becomes:

$$f = K*(n^4) + L*(n^2) + C$$

So we conclude that the algorithm has the worst case asymptotical complexity behaviour of $O(n^4)$. This is a very expensive algorithm. The reason is, we have exhaustively searched the second relation's table for each tuple of the first argument relation.

I. SECOND ANCESTRAL (san:R)

This operation takes a relation identifier and produces another relation which is the second ancestral (transitive closure) of the given relation.

The algorithm for the Hash-Incidence-Vector representation makes use of WARSHALL'S algorithm for bit matrices. [Ref. 4] Warshall's algorithm for incidence matrices can be defined as follows:

Input: A is the nXn incidence matrix of the given
relation, where "n" is the cardinality of the MEM set of
the given relation. (* This means the "row set" and the
"column set" of the incidence matrix are the same and are
the MEM set of the relation. *)

Output: R, the transitive closure of A, also as an
incidence matrix on the MEM set of the given relation.
Let k represent the column number and i represent the row
number. Let Rik denote the entry of the incidence matrix at
row i and column k. Let Ri be the i'th row of R for $0 < i < n+1$
and let V denote the OR operation on the rows of incidence
matrix.

Algorithm Transitive (input, output)

R - A

for k - 1 to n do

    for i - 1 to n do

        if Rik = 1 then Ri - Ri V Rk

    end do

end do

Note that Warshall's algorithm is defined for square matrixes
but that this is not the case for our incidence vector. Our
incidence vector originates from a different representation
of the incidence matrix which is not necessarily a square
matrix. According to our definition of the incidence vector,
given an incidence matrix, if we convert it to the incidence

102

vector representation, the row set of the incidence matrix corresponds to the LEM set of the incidence vector and the column set of the incidence matrix corresponds to the RIM set of the incidence vector. Note that the LEM set and the RIM set of a relation are not necessarily the same.

In order that the incidence vector of the transitive closure of a relation be different from the incidence vector of that relation, the LEM set and the RIM set of that relation must not be disjoint, otherwise the transitive closure of that relation has the same incidence vector as the original relation's incidence vector.

Now we have to modify Warshall's algorithm for our case. Note that our incidence vectors are more efficient in storage usage than the bit matrices used by the Warshall's algorithm, and still Warshall's algorithm works without an overhead in time.

The algorithm for Hash-Incidence-Vector representation is as follows:

1. Find the relation's record in the relation table by hashing with the given relation identifier to the relation table.

2. Follow the PFLM field of that record and find the first left member's record in the LHT.

3. Allocate a block of memory as large as:

$$(|LEM| * |RIM|)/C$$

103

where "C" is the memory word length, "|LEM|" is the cardinality of the LEM set of the given relation and "|RIM|" is the cardinality of the RIM set of the given relation. Record the beginning address of that memory block.

4.   Start from the beginning of the LEM set of the given relation.   For each LEM set record found by following the TASE links between the records, hash with the individual represented by that record into the RHT under the given relation's identifier.

a.   If the RIM set of that relation also contains that individual, get the index of the LEM set record which represents the individual in question, and call it "INDEX". Copy INDEX to INDEX+|RIM|-1 bits of the incidence vector of the relation to the corresponding bits of new incidence vector.

b.   Otherwise set a pointer to the LEM record of that individual and put it into a temporary array of type pointer, get the index of that record and call it "INDEX" and put zeros into the INDEX to INDEX+|RIM|-1 bits of the new incidence vector.

5.   Repeat step 4 until the LEM set of the relation is exhausted.

6.   Start from the beginning of new incidence vector and execute the loop below on the new incidence vector. (* NOTATION: In the algorithm segment below, "VECTOR[i,j]"

104

means the incidence vector location corresponding to the LEM
set individual which is associated with the index i and the
RIM set individual which is associated with index j, the
"VECTOR[i,j] to VECTOR[i,k]" means the cluster of incidence
vector entries (bits) beginning with the entry VECTOR[i,j]
and ending with the entry VECTOR[i,k]. *)

```
    for count1 = 1 to |LEM|*|RIM| by |RIM| do
      for count2 = 1 to |RIM| do
        if VECTOR[count1, count2] = 1 then
          VECTOR[count1, 1] to VECTOR[count1, |RIM|] =
            VECTOR[count2, 1] to VECTOR[count2, |RIM|] V
              VECTOR[count1, 1] to VECTOR[count1, |RIM|]
        end if
      end do
    end do
```

7.   Start from the beginning of the temporary pointer
array and find the record of each left member that is not
present in the RIM set of the relation by following the
pointers in turn.   For each record found in this manner
extract the index of that record, call it INDEX and OR the
INDEX to INDEX+|RIM|-1 bits of the original incidence vector
with the corresponding bits of the new incidence vector.

8. Make separate copies of the LEM set and the RIM set of original relation in the LHT and in the RHT respectively under the new relation identifier "san:R" as was done in the previous algorithms.

9. Hash to the relation table with the new relation's identifier and establish its record. Put the pointers to the new LEM set and new RIM set into the PFLM and the PFRM fields of that record respectively. Copy the |LEM| and |RIM| fields of the original relation's record into the corresponding fields of the new relation's record. Put the beginning address of the new incidence vector into the "base" field of that record.

Now we will do the worst case asymptotical complexity analysis of this algorithm.

We see that the algorithm is not as costly as it is expected to be. The worst case complexity function of this algorithm can be written as:

$$f = L*n*(n/C) + M*m*n*(n/C) + N*n + T*M + D$$

where:

   m = The cardinality of the LEM set of the given relation.
   n = The cardinality of the RIM set of the given relation.
   C = Memory word length.
   L = The constant number of memory references made while copying or OR'ing the clusters of incidence vector bits for each LEM set individual in steps 4, 5 and 7.

106

M = The constant number of memory references made for each iteration of outermost "for" loop in step 6.

N = The constant number of memory references made while copying each RIM set individual in step 8.

T = The constant number of memory references made while copying each LEM set record in step 8.

D = The constant number of memory references made by the remaining steps.

In the above function the first term corresponds to steps 4, 5 and 7, the second term corresponds to step 6, the third and fourth terms corresponds to step 8, and the last term corresponds to the remaining steps of the algorithm.

In writing the above complexity function we assumed that in the worst case the temporary pointer array will be empty because in the worst case the OR operations done in step 6 should be maximum. That means when we first make the separate copy of the original relation's incidence vector (which is the primitive form of the new relation's incidence vector); in the worst case it should consist of all one's.

Let $m=n$, $L/C = H$, $N+T = S$ and $M/C = I$, then the above function becomes:

$f = I*(n^3) + H*(n^2) + S*n + D$

We conclude that the worst case asymptotical time complexity behaviour of this algorithm is $O(n^3)$. In fact the algorithm becomes an order three algorithm if $n > C$, otherwise it can

107

be viewed as an order two algorithm, since when n > C we have to make (n/C) memory references for each OR operation. Again making use of pipelining and cache memory is advantageous in this case.

Now we will define the algorithm for the table representation. We will again use SCHT mechanism for this algorithm. The algorithm is as follows:

1. Call the algorithm "mem", compute the cardinality of the resulting MEM set of the relation and record it.

2. Start from the beginning of the given relation's table. For each right individual found by following the links between the records and by extracting the right individual represented by the "right" field of each record, hash into the SCHT with that individual, create an SCHT record for that individual and connect it directly to the SCHT entry found. In the case of a collision, use the rehashing technique. Create a SCHT record for the left individual of the current tuple and link the record of the right individual to that record by its collision link. (* Note that the SCHT records corresponding to the left individuals are not connected to the SCHT entries. *) If after hashing with the right individual it is found out that a record of that right individual is already present, then add the record of the current left individual to the end of the bucket connected to this right individual's record. Link the right individuals'

108

records to each other by their TASE links. Mark the beginning of the resulting linked list with the pointer P. (* As a result of step 2, the relation is represented in the SCHT without repetitions of the right individuals. All the SCHT records of the left individuals that are related with a right individual have been established in a bucket connected to the record of this right individual. *)

   3.   For I = 1 to |MEM| by 1 do

      a.   Start from the beginning of the linked list structure which connects the right individuals' records in the SCHT. By following the pointer P proceed in this linked list structure record by record. For each record found in this manner do step 3b.

      b.   Follow the pointer found in the collision link of the current right individual's record and find the left individual's record (which is in relation with the current right individual).

      c.   Extract the individual represented by the record found in step 3b; hash with this individual to the SCHT. If this individual is represented by an SCHT record which is directly connected to the SCHT entry found, follow the collision link of the record found and find the SCHT record connected to this record.

      d.   Extract the individual represented by the record found in step 3c. Search for this record in the bucket

connected to the right individual's SCHT record found in step 3a. If this individual is not represented by a record in this bucket; create a SCHT record for this individual and add it to the end of the bucket.

      e.   Proceed in the bucket connected to the record found in step 3c and for each record found do step 3c.

      f.   After the bucket is exhausted do steps 3c through 3e for the next record of the bucket connected to the SCHT record found in step 3b. (* After the execution of the above steps, the transitive closure of the given relation appears in the SCHT in the adjacency list representation. Next it has to be converted to the table representation. *)

    4.   Construct the table representation of the transitive closure of the relation by looking at the arrangement of the records of the individuals in SCHT (as was done in the step 5 of the algorithm given for the complement operation on relations).

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

Before going into the detail of the complexity function, we have to define what would be the worst case. In the worst case the relation is a universal relation on its MEM set. In that case the buckets constructed in SCHT have a common length, which is equal to the cardinality of the members set (MEM) of the relation. Let this length be "n", which means

110

the cardinalities of the MEM, RIM and LEM sets of the relation are all equal to n. So in step 3 we make a number of memory references proportional to n for each record found in step 3e and we make a number of memory references proportional to the square of n in step 3f by repeating step 3e n times. In the same way we make a number of memory references proportional to the cube of n in step 3a by repeating step 3f n times for each right member of the relation. Since the step 3 also iterates n times; in step 3 we make a number of memory references proportional to (n^4). So we write the worst case complexity function of this algorithm as follows:

$f = K*(n^4) + L*(n^2) + D$

where the first term corresponds to step 3, the second term corresponds to step 1, 2 and 4, and the last term corresponds to the number of memory references made by the housekeeping operations. The constant K represents the constant number of memory references made for each iteration of step 3. The constant L represents the averaged number of memory references made for each tuple of the original and/or the resulting relation (whichever is larger) in step 1, 2 and 4.

As can be seen the algorithm is a very costly algorithm. The high cost of this algorithm is caused by:

1. Repeated execution of relative product operation on the relation itself and on each intermediate relation found.

111

2.   Removing duplicate tuples from the intermediate relations.

This algorithm utilizes the SCHT mechanism for removing the duplicate tuples from the intermediate relations.

Suppose we defined the algorithm in a straight forward manner; i.e., the algorithm obtains the transitive closure by first getting the relative product of the relation with itself (R^2) and taking the union of the resulting relation and the original relation to obtain the next intermediate relation, and so on. If we do not remove duplicate tuples from the intermediate relations, in the worst case defined above each intermediate relation's size becomes two times greater than the previous intermediate relation's size, and the algorithm becomes an $O(2^n)$ algorithm automatically. On the other hand if we remove duplicate tuples from each intermediate relation without using SCHT mechanism, in the worst case defined above an operation of this kind has an $O(p^2)$ algorithm (where "p" is the relation size). If we accept the cardinality of the MEM set of the relation as a measure, the algorithm may be viewed as $O(n^4)$ algorithm. We can define an $o(n^5)$ algorithm for the second technique described above. Because for each step of that algorithm we will have to execute the algorithm that removes duplicate tuples, the resulting algorithm becomes an $O(n^9)$ algorithm, so our previous algorithm can be viewed as a relatively efficient

112

algorithm for obtaining the transitive closure on the table
representation, but it is so expensive that it is not
feasible to implement it at all.

III.   ANALYSIS OF INTENSIONAL ALGORITHMS

In this study we will try to find out efficient ways to represent intermediate relations and sets that result from relational operations and set operations. Earlier when we focused on the extensional representation techniques, we explicitly constructed the representation structures for the intermediate relations and sets in the memory. What we will try to do now is, not to represent intermediate relations and sets in the memory explicitly and still be able to execute the relational expressions.

In order to do the relational operations and set operations without explicitly representing the relations or sets resulting from those operations, it requires the complete establishment of the relational language's syntax. After the syntax is established we can decide on the suitable compiler or interpreter design and we can compile or interpret the source expression so that the code which does the relational operations and set operations without constructing the extensional representation structure of the resulting relation or set can be produced and/or executed. Because the syntax of the language is subject to changes, we will not go into the compiling or interpreting issues in

114

detail; instead we will define the algorithms for the code that does the relational operations and set operations intensionally.

Now we have to define what is the basic schema we have in mind. We assume that the relational language expression is fully parenthesized or there exists a default convention (left to right/right to left) which causes the expression to be parsed as if it is parenthesized by using this convention. The recursive descent parser parses the expression and for each pair of operators found in this manner calls the appropriate code that we will give the algorithm for. So we directly execute the source expression while parsing. Example:

Suppose the parser is to parse the expression below,

((R-S) | (T&U)):x

The scanner finds the "|" and ":" operators first, and passes the tokens to the parser. The parser then calls the routine defined for the "|" and ":" operator pair. This routine directs the scanner for finding the operands and the argument, then the scanner returns the token for the operator detected in the first operand relation (R-S) which is the token for "-". The routine attaches the appropriate tokens and argument to this token using the argument given to it and calls the routine defined for the resulting operator pair (In this case the routine associated with the "-" and ":"

operator pair is invoked with the argument being "x" and the operand relations being the R and S). If this routine does not return a valid individual, the same thing is done for the second operand (composite) relation. If an individual is obtained by calling any of those routines, this individual is either returned to the caller or sent to the output.

Now we have to explain this mechanism in general. We can view the parser as the collection of routines. After the first operator pair is found the related routine is called, and this routine gets the tokens it needs by directing the scanner; it attaches the appropriate tokens to those tokens and calls the routines related to the resulting operator pairs with the appropriate arguments possibly obtained from its own argument. Each routine called may do the same thing by directing the scanner to get additional tokens and relation identifiers from the source code. This process continues until the primitive operations can be done on the extensionally or intensionally represented relations and sets, then each routine returns the result to its caller, the caller performs the necessary evaluations on the results that are obtained by calling other routines, and returns the result obtained to its caller. This process continues until the final result is sent to the output.

As can be seen in the above example, each routine knows what it is doing and mechanically does its job by calling other routines defined for other operator pairs. Note that a routine may call itself again in some depth of calling chain. This process stops when the primitive operations can be executed on the extensionally represented relations and sets. In fact, some primitive operations can also be executed on the intensionally represented relations, which will be discussed later.

So in summary, the process defined in the above example is aimed at reducing the initial compound relational expression into easily manageable primitive relational operations and the membership tests on the extensionally or intensionally represented relations and sets. Note that the above process is just a particular case of the use of the algorithms that we will define; these algorithms can be adapted to the other implementation schemas. So in defining our algorithms we will not go into the detail of the implementation technique we proposed.

Because there are no less than 136 possible permutation of operator pairs, if we create a tiny routine for each case, we may cause trashing. In fact we do not have to introduce that many routines into the system; first of all most of the 136 routines do very simple, well defined reductions. In addition the interpreter we have in mind directly executes

the source code. Under those considerations we may preprocess the source code by directly making simple reductions on the source code. This mechanism simply takes a part of the source code and replaces this part with a reduced expression whenever it finds a part of the source code which is reducable with the rules in hand. So executing this text substitution mechanism separately, leaves us less work in interpreting the source code. But this is not enough to make the system simple and compact. So we take advantage of the regularities in the operations and generalize some operations so that we do not have to define all possible operator pairs involving that operation. That leaves us a total 62 special cases or in other words operator pairs to be specially treated or requiring an algorithm.

The second issue in that kind of mechanism is, to generate the individuals of an intensionally represented set one at a time as they are requested. This becomes necessary in some relational operations that require the individuals of the argument set in order to accomplish their job. In some cases we want to learn only, if a given individual is in an intensionally represented set or not. In this case we can do membership test with less cost, although some configurations of composite, intensionally represented sets still forces us to produce the individuals of that intensionally represented sets explicitly in order to do the membership test. Thus in

118

some operations, (like the "Image" operation) we will have to define algorithm for both the membership test and the production of the individuals of the sets produced.

In defining that mechanism, first we wrote down all the possible operator pairs, we separated out the ones that can be handled in the preprocessing phase, then we generalized some of the operations and treated the remaining operator pairs as special cases.

In the following sections we will explain some concepts, such as generalization of operations and the structure of the system, further.

A. PREPROCESSING

The complete listing of preprocessing rules is given in Appendix C. In this section we will explain how the mechanism works by giving examples.

The preprocessing mechanism scans the source code and, whenever it finds a pattern matching one of the rules that it knows about, makes the necessary modifications to the source code. To make this clear we will give an example. Example:

Suppose our source code contains the segment of code below:

$((RS) - (final:T))c$

The preprocessing mechanism finds the operators (by counting the parenthesis) "c" (converse) and "-", and the rule defined

119

for this case is applied. In this case the applicable rule
is:

$(R-S)c = Rc - Sc$

So the preprocessing mechanism makes a seperate copy of the
character string, "(RS)", concatanates the character "c" to
it, takes the character string identifying the second operand
relation which is "(final:T)", and concatanates it with the
character "c". The preprocessor inserts the character "-"
between the two character strings obtained and attaches the
parenthesis to both ends of the resulting character string.
Hence the resulting character string replacing the original
segment of source code becomes:

$((RS)c - (final:T)c)$

So, as can be seen the mechanism is fairly mechanized and
simple. All the preprocessing mechanism has to do is to
count the number of paranthesis and find the operator pairs
for which the preprocessing rules are defined. Then it makes
the necessary modifications in the source code. Note that
the preprocessing program has almost no intelligence; on the
contrary, it does the reduction mechanically.

We want to emphasize that the preprocessing program does
not work only once and can be implemented so that it makes
more than one pass on the source code in order to apply
preprocessing rules further if the first pass yields a source
code that can be further preprocessed, or it may be executed

120

on a segment of source code upon request of any routine of
the interpreter.

B. GENERALIZATION OF OPERATORS

We can divide the relational operators into two groups:

1. The operators that construct the new relations from
the operand relations.

2. The operators that construct sets.

We list the operators of the first kind as follows:

    a. R&S (Relation intersection).

    b. R|S (Relation union).

    c. R-S (Relation difference).

    d. Rc (Converse of a relation).

    e. non:R (Complement of a relation).

    f. R||S (Parallel application).

    g. R#S (Dual application).

    h. fan:R (First ancestral of a relation).

    i. san:R (Second ancestral of a relation).

    j. R::G (Meta application).

    k. R:x (Function application).

    l. R\C (Relation right restricted to the set C).

    m. C/R (Relation left restricted to the set C.)

    n. C/R\C (Relation restricted to the set C).

    o. final:R (Final members of relation R).

In the above listing we can select any of the composite
relations and substitute another composite relation into any

121

operand and/or argument of it. In this manner we can combine
any number of relations. As an example we will make a few
substitutions in order to obtain a new composite relation out
of the other composite relations. Suppose we selected item
(i) which is "san:R" and substituted the relation "R&S" into
the argument R, we obtain:

san:(R&S)

and suppose we further substituted "C/R" into the R and
"R||S" into the S in the above composite relation. We
obtain:

san:((C/R)&(R||S))

Now we have to list down the relational operations that
construct s⁻ts. These operations are given below:

a.   R:x (Function application).

b.   lem:R (Left members set of a relation).

c.   rim:R (Right members set of relation R).

d.   R!:x (Image).

e.   unimg:R (Unit image).

f.   unimg':R (Unit coimage).

g.   init:R (Initial members of relation R).

Definition: A composite set is a kind of set which is
expressed in terms of composite relations and relational
operations.

We can substitute any composite relation into the R in
the above cases. (Arguments shown as "x" means that the

122

argument is individual). Note that the "Function Application" operation is included in both cases, because it may produce an individual which can be a set, a relation, an integer, a real, a character string or a bit string. We can combine the above composite sets by using the set operations "Union", "intersection" and "difference". By attaching the negation sign in front of each of the above composite sets we get the complements of those sets which are again viewed as composite sets. The user-defined (primitive) extensionally or intensionally represented sets can participate in the set operations with the composite sets. So we can define an infinite number of composite sets.

As we mentioned earlier, considering each of the possible pair of operators causes the system to be very complex, so we generalize the operations of the first kind in order to recover from doing that. In order to make this concept clear we will take a specific case and explain what we mean by generalization.

As a specific composite relation let's take the relation "fan:R". We can substitute 14 other relational operations (except meta application) into the R including "fan:R" itself. Because each of the composite relations constructed in this manner can participate in the operations of the second kind or the first kind, we would be unable to cover all the cases that may be defined by the programmer. So in

123

the generalization of "fan:R" we will define an algorithm for
"fan:R" for each of the operations of the second kind. For
example we will define an algorithm for the operation,
(fan:R)!: which is constructed by using ! and fan operations.
The ! operation is the kind of operation that constructs a
composite set and the fan operation is the kind of operation
which constructs the composite relation. That means the !
operation is of the second kind and fan operation is of the
first kind. These algorithms will be defined in terms of the
operations of the second kind working on the operand relation
R. So an algorithm defined for the operator pair <fan,!> or
in other words for the operation (fan:R)!: applies the
operations of the second kind to the relation substituted for
"R". Because the algorithms of these operations are also
defined for this relation, the ! (image) operation on "fan:R"
can be done with no confusion. Note that the relation
substituted for "R" may be another composite relation
constructed from the operations of the first kind and the
same rule applies to this relation as it was in the relation
"fan:R" case. This rule is applied until the operations of
the second kind can be done on the extensionally or
intensionally represented user defined relation. Example:

Suppose we have the relational expression:

(fan:(san:R))!:x

The prime operators in this expression are the left-most fan operator and the ! operator. In this case the algorithm defined for (fan:R)!: operation is invoked. Suppose further, this algorithm uses unimg operation (which is of the second kind) on the operand relation (which is the san:R) so the operation to be performed is unimg:(san:R) and the system has an algorithm defined for the <unimg,san> operator pair. Note that the algorithm defined for the <fan,!> operator pair does not care about the kind of composite operand relation and no matter how complex this operand relation is, it simply applies the operations of the second kind to the operand relation (in this case san:R) in order to do its job. That is what we mean by generalization.

C. THE ALGORITHMS FOR GENERATING THE INDIVIDUALS OF
    COMPOSITE SETS

In this chapter we will explain in detail how we produce the individuals of the intensionally represented intermediate sets. In addition to that we may want to test if a given individual is in an intensionally represented set. As long as we are able to produce all the individuals in intensional sets, checking for membership is trivial. But producing all the individuals in these kinds of sets is a costly operation and in some cases we can do the membership test with less cost. On the other hand, we can not restrict ourselves to only membership tests in order to work around the costly

125

production mechanism because some of the relational operations expect each individual of a given argument set in order to accomplish its job. We will need to produce the individuals of intensionally represented intermediate sets especially in the "!" operation and in the operations on the complement of a relation.

In the production of the individuals of an intensionally represented set, defining one primitive function is very useful. We will call this primitive function, "Force". When we apply this function to an intensionally represented set for the first time, it returns the first individual of this set. Repeated applications of this function to this set will return the second, the third individuals in turn, and so on. Thus the function must set up break points between the production of the individuals of the intensionally represented sets. At those break points we examine and evaluate the individual returned or check if it satisfies some condition. Another objective of the "Force" primitive is that, if we are doing the membership test by producing the individuals of an intensionally represented set by comparing the given individual with each individual produced, as soon as we find a match, we can quit producing the individuals, thus saving ourselves from producing all the individuals of the intensionally represented set in question.

Implementation of the "Force" primitive is fairly complex. We will show that there exists at least one way to implement the "Force" primitive. Our force primitive, in fact does everything in the production of the individuals of an intensionally represented intermediate set.

In the discussion of the types of the relational operations, we defined two types of relational operations and we stated that, we can combine the two kinds of relational operations to obtain composite sets. In addition we can use the set operations as the resulting composite sets being the operand sets in order to obtain other composite sets. We can easily define the role of the set operations in the production of the individuals in general (i.e., we can generalize the set operations). That leaves us all the distinct configurations of composite sets that can be created from the operations of the first kind and the operations of the second kind by substituting the operations of the first kind in the argument and/or operand relation (R) of the operations of the second kind. Because "init:R" is defined in terms of "lem:R" and "rim:R", and we will define the algorithms involving the function application operation in Chapter E, in this chapter we will only focus on the operations below:

lem:R

rim:R

```
R! :C
unimg:R
unimg':R
```

which are of the second kind. In the same manner, we will
generalize the operations "Right restriction", "Left
restriction" and "Restriction" which are of the first kind.
As before, the "Function application" operation returns a
composite relation of the first kind so there is also no need
to treat this operation as a special case. Those reductions
leave us the remaining operations of the first kind which are
to be combined with the operations of the second kind.

We will define the "Force" primitive as a recursive
function which includes less than 30 cases defined for the
distinct permutations of the first and second kind of
operations. Because we preprocess the source code many of
the permutations are reduced to the other permutations that
we will be defining the algorithms for. The "Force" function
includes a big case statement in which each case refers to a
particular permutation (operator pair) that will be treated
specifically. It takes the character string that identifies
the composite set, and extracts the operator pair of this
composite set. It identifies the argument (if there is any)
and invokes its appropriate case in the case statement. This
segment of case statement calls the function "Force"
recursively by subdividing the original composite set and

128

expressing it in terms of operations of the second kind on the operand relations/relation. That means the particular case of the case statement mechanically finds the operand relations in the composite set expression, creates other composite sets by using those relations and the argument (if there is any) and forces the composite sets by calling the function "Force" recursively with those composite sets as the arguments. This recursion continues until an operation of the second kind can be performed on an extensionally represented relation or set. Note that "Force" has to recognize when it finds a primitive extensionally represented relation or set. That can be done by hashing into the relation table or the set table and by checking if a record of the relation/set exists in the relation-table/set-table.

We stated that the "Force" function returns one individual at a time. This feature makes the program extremely complex. Now we will explain how that mechanism works. We know that we get the individual to be returned from an extensionally represented relation or set and this is the stopping condition for our recursion. Further, after returning an individual, we have to remember where we left in order to respond to a subsequent reference to the function "Force" related to the same argument composite set in the same context. In fact memorizing where we left off is

necessary at a very low level, i.e., in the operations on the extensionally represented relations and sets. Example:

Suppose we are forcing the composite set, rim:(R-S) where R and S are assumed to be primitive, extensionally represented relations. According to the algorithm defined for the operator pair, <rim,-> (which will be given later) we first force lem:R-lem:S and get the first individual of lem:R-lem:S say "y". We force the composite set unimg':R:y next, so we get the first individual and return it. Suppose now the composite set rim:(R-S) is forced again. The individual to be returned is the next individual of the composite set unimg':R:y. So we have to remember where we left off in the unimg':R operation.

In this example, we can solve this recognition problem by setting a global pointer to the record of the next individual to be returned in the RIM set of R. Obviously this does not solve all the problems, but we have to observe one fact here: Suppose our original composite set has been a part of another composite set and the first individual we produced had to be tested against the unimg':R:y (composite set) in another context. That means we are in the situation that we have two same operations (unimg':R:y) being used for different purposes in the same expression. An example for this case is given below:

(unimg':R:y)/G!:(unimg':R:y)

In execution of this expression the individual obtained by forcing the composite set:

G!:(unimg':R:y)

is tested if it is in the composite set (which is on the left of G) unimg':R:y or not (as a result of restriction operation). So the unimg':R:y operation is performed for two different purposes and we perform each operation while we still save the state of the other. The second reference is faced with the strange fact that, even though it is unrelated with forcing the (unimg':R:) operation for obtaining the second element of the composite set unimg':R:y it obtains the second individual of the right-most composite set unimg':R:y, instead of the first individual of the left-most composite set unimg':R:y.

Based on the above facts we have defined the mechanism below in order to make this work.

Definition: A "high level composite set" is a composite set which is either defined in the source code or created by an internal mechanism other than the function "Force". The mechanism has the below structure:

1. Memorizing the state is done in these operations when they work on the extensionally represented relations:

    a.   Image operation.

    b.   unimg operation.

    c.   unimg' operation.

d.   left members operation.

e.   right members operation.

2.   Each high level composite set is associated with a
global count, when it is forced for the first time, and a
system-wide association table is maintained which relates the
high level composite sets with their global variables. Each
global count associated with a high level composite set
begins from a biased value so that the values of the global
counts associated with various high level composite sets are
restricted to the particular intervals.*

3.   The system has a global hash table, namely MHASH
table (Memory Hash Table) which is used to save various
pointers.

4.   In each level, when a composite set is created, an
integer taken from the global count associated with the high
level composite set (from which this sub-composite set is
originated) is saved. When a pointer is set to a record at
the lowest level, this pointer is saved in the MHASH table
with the current integer being the identifier with which we
hash into the MHASH table. (This allows us to save different
pointers for the same operation and the same extensional
relation or set. Hence we properly distinguish between the

---

*Because the system will have a limit on the number of
high level composite sets this interval can be defined in the
implementation phase.

132

same operations that request the same things from the same extensional relations but have different originations).

5. When a new composite set is created, the pointer to the character string representing this composite set is saved in the MHASH table with the current integer being the identifier. (This allows us to avoid creating the same sub-composite sets redundantly in the subsequent execution of the "Force" function related to the same high level composite set and originated from the same context).

6. If there is no individual remaining to be returned in a primitive relation or a set, we associate 0 with the current integer in the MHASH table, instead of a pointer, and we return "nil" to the caller (which is the function "Force" in itself).

7. After each force operation on a high level composite set, the global count associated with this high level composite set is reset to 0.

So, suppose we forced a composite set, and we got the first individual of this composite set; if we force this set again, the function goes and makes the same calls and goes through the same kind of counting mechanism. When it hits a primitive relation, it hashes with the current integer into the hash table and looks up the pointer. If it is not zero, it returns the next individual obtained from the LEM set or the RIM set of the primitive relation or the extensionally

133

represented primitive set, by advancing the pointer identified by this integer, and re-establishes this pointer in the hash table with the same integer being the identifier. If there is no pointer but 0, it returns "nil" to the caller, in which case the caller (the function "Force" itself) proceeds to find the next primitive composite set to continue to produce its individuals, if any remain. (Note that we are always mentioning the function "Force" because it calls itself recursively with different arguments).

So when we force a composite set repeatedly the function "Force" remembers its last state and returns the next individual of the composite set without any confusion. Because the calling chain has to reoccur and the counting mechanism always follows the same procedure, we do not get integers different from the ones established in the first pass.

So this mechanism uses random coding principles, and the hashing is done through a counting process.

Now we will define the algorithms for all possible cases that should be included in the big case statement of the function "Force". As we explained before, many kinds of composite sets can be reduced to the other types of composite sets or primitive ones in the preprocessing phase. By primitive we mean that the relations and sets from which the composite set is constructed are represented extensionally.

134

So our algorithms will refer to the kind of composite sets that should be treated specifically.

In these algorithms we will try to emphasize the fact that the individuals are returned one at a time, but in some cases it is more descriptive, if we explain an algorithm as if it is being forced repeatedly. In addition we will not repeatedly mention the state saving mechanism which saves the pointers in the MHASH table, with the identifiers being the integers taken from the global count. So the reader should always think that .pa the individuals are returned one at a time and the state saving mechanism works as it is supposed to do.

In these algorithms we will use the LHT as the SCHT whenever it is possible to produce the same individuals repeatedly as a result of forcing a composite set repeatedly. Hence at any point in execution we remember which of the individuals of the composite set being forced have been produced as a result of previous force operations, and we do not produce them again. This mechanism is also useful in the intermediate operations done in some algorithms and reduces the time complexity of some algorithms in some cases.

In this mechanism we construct a set in the LHT which contains the LHT records of the individuals that we want to save temporarily. After no forcing operation can be made to a composite set we may return the linked list structures of

135

all sets constructed in the LHT for this composite set to a storage pool (which may be implemented as a stack), so that the subsequent record allocations can be made from the storage pool; if the storage pool is empty then the records are allocated from the heap. If the storage pool size exceeds some previously defined limit, we dispose some number of records from the storage pool. As a consequence, this kind of mechanism does not use up a lot of memory resources and we recover from doing our operations redundantly.

In this mechanism we will again use the integers taken from the global count as identifiers; because we are distinguishing the sets constructed in the LHT with the relation identifiers, these integers will be treated as if they are relation identifiers. In order to return the linked list structures of these sets to the storage pool, a separate table should be maintained that includes the identifier of the set (integer) and its origination (i.e., which composite set being forced caused that set to be created).[*]

Suppose we did not define this mechanism, and for each individual produced by forcing a composite set we did some complex operation. Then if the same individual is produced

----
[*]The reader should not confuse these two usages of the integers as identifiers. In summary, we use the integers taken from the global count for saving the state of the "Force" function and for designating the intermediate, temporary sets in the LHT. As long as we allocate the same integers at the same points each time a high level composite set is forced, no problem arises.

two times, we will do this complex operation redundantly when the individual is produced for the second time.

Image operation on the extensionally represented relation :

We will need to modify the algorithm for image operation that we defined previously because in this case it has to produce the individuals of the resulting set one at a time.

The new algorithm for "Image" operation makes use of the "Unit image" operation for which we will be defining the algorithm for later. The algorithm is as follows:

1. Force the argument set C.

2. Get the individual returned say "x".

3. Force the primitive composite set unimg:R:x; take the individual returned, say "y".

4. Take the next integer from the global count; hash with this individual into the LHT with the integer obtained above being the relation identifier. If this individual has a record in the LHT under this relation identifier go to step 3 in order to continue with the next individual of the composite set unimg:R:x, otherwise do step 5.

5. Establish the record of this individual in the LHT under the relation identifier (integer) obtained in step 4; return this individual as the result.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

137

As can be seen we executed the "Unit image" algorithm for each individual of the set C. So we made a number of memory references proportional to "n" for each individual of the set C, where "n" is the cardinality of the LEM set of the extensionally represented relation. Suppose the cardinality of the set C is equal to "n", then the time complexity behaviour of the algorithm is $O(n^2)$. But suppose the cardinality of the set C is equal to the square of "n". This time the time complexity behaviour of this algorithm becomes $O(n^3)$. On the other hand the cardinality of the set C may even be 1 (singlteon set); in that case the time complexity behaviour of the algorithm becomes $O(n)$, which is the same as the asymptotical time complexity of the "unimg" algorithm that will be given later. So we conclude that the time complexity behaviour of the algorithm is strongly dependent on the cardinality of the argument set.

(R - S)!:C

The algorithm for this case is as follows:

1. Force the set C.

2. Get the individual returned. Call this individual "x".

3. Force unimg:R:x; get the individual returned. Call this individual "y". Get the next integer from the global count, hash into the LHT with this integer being the relation identifier, check if there is a record for "y" in the LHT

under this relation identifier, and if so do step 3-a, else do step 3-b.

a. Repeat step 3 in order to continue with the next individual of the unimg:R:x.

b. Force the unimg':S:y repeatedly and get the individuals one at a time. If any of those individuals is the same as "x", or if "y" is not in the lem:S (i.e., the (unimg':S:) operation is not applicable) quit forcing the unimg':S:y and go to step 3 in order to continue with the next individual of the composite set unimg:R:x. Else do step 3-c.

c. Establish the record of "y" in the LHT with the integer obtained in step 3 being the relation identifier and return "y".

4. If as a result of the subsequent force operations no individuals of the unimg:R:x remains to be returned, force the set C in order to get the next element of it and continue from step 2 as the algorithm is forced further.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

Assuming we will force the above algorithm until no individual of the set (R-S)!:C remains to be produced, and assuming in the worst case the relations R and S are disjoint, we make the number of memory references proportional to "n" for each individual of the set C, and for

each "y" obtained in this manner, because of the relations
are disjoint, we make the number of memory references
proportional to "n" where "n" is assumed to be the
cardinality of the both lem:R and rim:S.   Let's assume the
cardinality of the argument set is also "n", so it is clear
that we are making a number of memory references proportional
to the cube of "n".   This leads us to the fact that, if the
operand relations are represented extensionally the algorithm
behaves as an $O(n^3)$ algorithm; if the operand relations are
composite relations the cost of this algorithm increases
significantly depending on the cost of the "unimg" and
"unimg'" operations on those composite relations.

(R‡S)!:C

The algorithm for this case is as follows:

1.   Force the set C.

2.   Get the individuals of the set C one at a time.

3. For each individual found in the above manner (which
is necessarily a pair, otherwise the operation is
undefined), extract the left individual of this pair, apply
the relation R to this individual, and in the same manner
apply the relation S to the right individual of the pair.   If
both application operations return individuals, construct a
pair relation with these individuals and return the pointer
to the record of this pair.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The algorithm has the same worst case asmptotical complexity behaviour as the algorithm defined for the composite set (R#S)!:C and the same argument applies.

(RS)!:C

The algorithm for this case can be defined as follows:

1. Force the set C.

2. Get the individual returned and call it "x".

3. Force unimg:S:x] Get the individual returned and call it "y".

4. Take the next integer from the global count and hash into the LHT with this integer being the relation identifier and check if "y" has a record in the LHT under this relation identifier, if so go to step 3 in order to continue with the next individual of the unimg:S:x or if no individual of the unimg:S:x remains to be produced go to step 1 in order to continue with the next individual of the argument set C. Otherwise do step 5.

5. Establish the record of "y" in the LHT with the relation identifier being the integer taken from the global count in step 4, and force unimg:R:y. Take the individual returned and do step 6.

6. Take the next integer from the global count, hash into the LHT with this integer being the relation identifier,

check if there exists a record for this individual in the LHT with this integer being the relation identifier. If so go to step 5 in order to continue with the next individual of the unimg:R:y, and if no individual remains to be returned from unimg:R:y, go to step 4 in order to continue with the next individual of the unimg:S:x or the argument set C. Otherwise establish the record of the individual obtained from unimg:R:y in the LHT with the relation identifier being the integer taken from the global count in step 6 and return this individual.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

Suppose the relations R and S are extensionally represented relations. If we force the above algorithm repeatedly until no individual remains to be produced by forcing unimg:S:x for each individual "x" of the set C, we make a number of memory references proportional to the square of "n", where "n" is assumed to be the cardinality of both the argument set C and the lem:S. In the worst case we produce the complete LEM set of the relation S without any repetition of the individuals, because we save the individuals that have been produced until now by establishing each individual's record in the LHT. So in the second part (step 5) of the algorithm we force the unimg:R:y, at most "n" times, for each of the "n" y's, where "n" is the cardinality

142

of the lem:S. So in that part of the algorithm we make a number of memory references proportional to the square of "n", where the cardinality of the lem:R is also assumed to be "n". Under those conditions the execution of the above algorithm is effectively the same as the execution of two $O(n^2)$ algorithms sequentially. That means our algorithm has the worst case asymptotical time complexity behavior of $O(n^2)$.

unimg:R:x (Where R is Represented Extensionally)

This can refer to the "unimg" operation on an extensionally represented relation. Note that in this case R should not be considered as a composite relation. The algorithm for this operation is essentially the same as the algorithm that we defined in the extensional representation analysis, but in this case we are not producing all the individuals of the resulting set at once. Instead, the first time this operation is forced we find the first individual that belongs to the resulting set and we advance the pointer to the next individual's record in the LEM set of the relation (if there is one). Then we take the next integer from the global count, and we establish the pointer in the hash table with this integer being the identifier. We then return the individual we found to the caller. If we exhausted the LEM set of the relation after repeated force operations, we save 0 in the MHASH table using the same

143

integer as the identifier. So the main difference is we return the individuals of the resulting set one at a time as this composite set is being forced.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The algorithm has the same asymptotical time complexity behaviour as the algorithm that we defined in the extensional representation analysis for this operation, but in that case it is forced repeatedly until no individual remains to be produced.

unimg:(RS):x

The algorithm for this case can be defined as follows:

1. Force the unimg:S:x.

2. Get the individual returned and call this individual "y".

3. Force the unimg:R:y; get the individual returned; take the next integer from the global count and hash into the LHT with the individual in question under the relation identifier (Integer) obtained above. If this individual does not have a record in this set, or if this set does not exist, establish the record of this individual in the LHT with the relation identifier being the integer obtained above and return this individual as the result. Else, if this set already exists (resulting from previous executions of this algorithm) and if this individual is represented by a record

in this set, quit with this individual and continue with the next individual of the composite set unimg:R:y. In the case no individual of this composite set remains to be produced go to step 1 in order to continue with the next individual of the composite set unimg:R:x. (*This step prevents us from producing the same individuals repeatedly*). Note that when we force this composite set a second time we get the next individual to be returned by the force operation on unimg:R:y if there remains an individual to be returned. Suppose there is no individual that remains to be returned. The algorithm continues with the next individual to be returned by the force operation on the composite set unimg:S:x and repeats steps 2, 3 and 4.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As can be seen, for each individual obtained by forcing the unimg:S:x we force the unimg:R:y, where "y" is the individual obtained by forcing the unimg:S:x. Assuming the relations R and S are extensionally represented relations and assuming we force the composite set, unimg:(RS):x until no individuals remain to be produced, we make a number of memory references proportional to the square of "n", where "n" is assumed to be the common cardinality of the LEM set of R and the LEM set of S. So the algorithm has the worst case asymptotical time complexity behaviour of $O(n^2)$.

145

lem:R (Where R is Represented Extensionally)

This case refers to the "lem:" operation on an extensionally represented relation. The algorithm for this case is as follows:

1. Hash to the relation table with the relation identifier; find the record of the relation.

2. Follow the PFIM field of this record and find the record of the first LEM set individual.

3. Advance the pointer to the next record; take the next integer from the global count and save this pointer in the MHASH table with this integer as the identifier.

4. If you are forced again, take the next integer from the global count, hash with this integer into the MHASH table, take the pointer stored with this integer as the identifier, decrement the global count, and repeat step 3.

As can be seen the algorithm returns one individual at a time as it is forced.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

In the worst case the algorithm is forced until no individual remains to be returned from the LEM set of the relation. This effectively corresponds to tracing through the LEM set of the relation. That means we make a number of memory references proportional to "n", where "n" is the cardinality of the LEM set of the relation. So we conclude

that the algorithm has the worst case asymptotical time complexity behaviour of O(n).

lem:(R-S)

The algorithm for this case is as follows:

1.   Force the rim:R.

2.   Get the individual returned and call this individual "y".

3.   Force the unimg:R:y.

4.   Get the individual returned and call it "z".

5.   Force the unimg':S:z

6.   Get the individual returned, if this individual is the same as "y", go to step 3 in order to continue with the next "z". If no individual remains to be produced from the set unimg:R:y then go to step 1 in order to continue with the next "y" (i.e., get prepared for continuing with the next individual of the rim:R in the case the algorithm is forced subsequently).   Otherwise do step 7.

7.   Take the next integer from the global count; hash into the LHT with this integer being the relation identifier. Check if the individual to be returned (current "y") has a record in the LHT under this identifier. If so go to step 3 or if no individual of the composite set unimg:R:y remains to be produced in step 3, go to step 5 and execute step 5 for the current z.   Else establish the record of this individual

147

with the integer taken from the global count being the relation identifier and return this individual.

In the above algorithm getting prepared for producing the next individual of the set rim:R is no more than erasing the pointers saved for the unimg:R:y and the unimg':S:z operations in the MHASH table. So a subsequent forcing operation will find that there does not exist any pointer saved for the unimg:R:y operation and will automatically force the set rim:R in order to get the next individual following the "y". Note that if a subsequent force occurs, and if there remains individuals in the set unimg':S:z, the algorithm begins from step 4, and produces the next individual of the unimg':S:z which is to be tested against the particular "y" in question. If no individual remains to be returned from the set unimg':S:z, then the algorithm begins from step 3, and produces the next "y" and so on.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

In the worst case the relations R and S may be disjoint and in this case, assuming we force the composite set lem:(R-S) until no individual remains to be returned, the structure of the algorithm effectively becomes similar to the three nested "for" loops. Assuming the cardinalities of the rim:R, lem:R and the rim:S are all equal to n, the operation makes a number of memory references proportional to the cube of n,

148

because we are making a number of memory references proportional to "n" for each individual of the composite set rim:R. By forcing the unimg:R:y for each individual obtained from this composite set we are making a number of memory references proportional to "n" by forcing the composite set unimg':S:z. So we conclude that the worst case asymptotical time complexity behaviour of this algorithm is $O(n^3)$.

lem(R#S)

The algorithm for this operation can be defined as follows:

1. Force the rim:R; get the individual returned.

2. Apply R and S to this individual. If both applications return individuals construct a pair relation out of the individuals returned. Else go to step 1.

3. Return the pointer to the record of that pair relation which is established in the relation table.

4. Do the same as above in the subsequent force operations on this algorithm by producing the individuals of the rim:R one at a time.

In the above algorithm the application operation on S may not always return an individual, because not all individuals of the rim:R are necessarily in the rim:S. In fact we should have produced the individuals of the intersection of the operand relations' RIM sets in step 1, but this would be a very costly operation by requiring the individuals of the RIM

149

set of either or both of the relation R and relation S to be saved temporarily. So we used the application operation's filtering property and we only produced the RIM set of relation R. Given an individual in the rim:R if this individual is not in the rim:S, application operation on S fails to produce an individual as a result, because, we need two individuals in order to construct a pair relation. In this case our algorithm continues with the next individual of the composite set rim:R.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

In this algorithm, assuming R and S are extensionally represented relations, we make a constant number of memory references for each individual of the rim:R. Because we have to produce each individual of the rim:R in order to produce all the LEM set individuals of the relation (R#S), we make a number of memory references proportional with "n", where "n" is the cardinality of the RIM set of R. So we conclude that the worst case asymptotical time complexity behaviour of this algorithm is $O(n)$.

lem:(R||S)

When we constructed the extensional representation structure of the relation "R||S" earlier, we created all the possible ordered pairs that can be created from the LEM set individuals of the operand relations in order to construct

the LEM set of this relation. So our algorithm does the same thing by constructing those pairs one at a time. The algorithm is as follows:

1. Force the lem:R; get the individual returned and call it "x".

2. Force the lem:S; get the individual returned and call it "y".

3. Construct a pair relation with "x" being the left member and "y" being the right member.

4. Return the pointer to the record of that pair which is established in the relation table.

5. In the subsequent force operations, continue to produce the individuals of the lem:S and pair each individual returned with "x". Return the pointers to the pairs one at a time.

6. When the lem:S is exhausted, produce the next individual of the lem:R and repeat the above steps for this individual as you are forced.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

Assuming the relations R and S are extensionally represented relations, and the algorithm is forced exhaustively, we produce all the individuals of the lem:S for each individual of the lem:R, so if the cardinalities of the lem:R and the lem:S are both equal to "n", we make the number

151

of memory references proportional to the square of "n". That means the algorithm has the worst case time complexity behaviour of $O(n^2)$.

lem:(RS)

The algorithm for this case can be defined as follows:

1. Force the lem:S; get the individual returned. Call this individual "x".

2. Force the unimg:R:x; get the individual returned; hash into the LHT with the current integer (taken from the global count) being the relation identifier; check if there is a record for this individual under this relation identifier; if so do step 2-a, else do step 2b.

a. Repeat step 2 by forcing the unimg:R:x again and taking the next individual.

b. Establish the record of this individual in the LHT with the relation identifier being the current integer taken from the global count (i.e., the integer taken from the global count in step 2). Return this individual as a result.

3. Repeat step 2 as the lem:(RS) is forced and as long as there remain individuals in the unimg:R:x.

4. If there does not remain any individual in the unimg:R:x, repeat steps 1 and 2 as the lem:(RS) is forced, by forcing the lem:S and obtaining the next individual from the lem:S.

In this algorithm we used the LHT as the SCHT for remembering the individuals produced until now and for not producing the same individuals repeatedly, because the sets produced by "Unit image" operations may not always be disjoint or they may be the same. If we did not do that we might produce the same individuals repeatedly and if a complex operation were being performed on each individual produced we might execute that complex operation redundantly.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

If we execute the above algorithm until no more individuals of the lem:(RS) remain to be produced, we produce the whole LEM set of the relation S and for each individual of that set we perform the "unit image" operation. Assuming R and S are extensionally represented relations, and lem:S and lem:R have the same cardinality "n", we make a number of memory references proportional to the square of "n". So the worst case asymptotical time complexity behaviour of this algorithm is $O(n^2)$. Note that this is true if and only if the relations R and S are extensionally represented relations. In the case they are composite relations the time complexity behaviour of the algorithm may change depending on the time complexity behaviour of the operations, "unimg:" and "lem:" on those composite relations.

153

rim:R

The algorithm for this operation is the same as the algorithm for the operation "lem:R", but in step 2 of this algorithm we have to follow the pointer found in the PFRM field of the relation's record instead of the PFLM field. Hence the asymptotical time complexity behaviour of this algorithm is the same as the asymptotical time complexity behaviour of the algorithm for the "lem:R".

rim:(R-S)

This algorithm does the reverse operation that we defined in the algorithm for lem:(R-S). The reader should make the substitutions below in the algorithm for the "lem:(R-S)" in order to define the algorithm for the "rim:(R-S)":

    lem:R ---------> rim:R

    unimg:R:y -----> unimg':R:y

    unimg':S:z ----> unimg:S:z

So the asymptotical time complexity behaviour of this algorithm is the same as the asymptotical time complexity behaviour of the algorithm for "lem:(R-S)".

rim:(R||S)

This algorithm is similar to the algorithm of the "lem:(R||S)". We have to make the substitutions below in the steps of the algorithm defined for "lem:(R||S)" in order to define the algorithm for "rim:(R||S)":

154

```
lem:R ---------> rim:R

lem:S ---------> rim:S
```

So, obviously, the asymptotical time complexity behaviour of this algorithm is the same as the asymptotical time complexity behaviour of the algorithm for "lem:(R||S)", because the algorithms for the "lem:X" and "rim:X" have the same asymptotical time complexity behaviour when "X" is an extensionally represented relation.

rim:(RS)

This algorithm is similar to the algorithm defined for "lem:(RS)". We have to make the substitutions below in the steps of the algorithm defined for the "lem:(RS)" in order to define the algorithm for "rim:(RS)":

```
lem:S ---------> rim:R

unimg:R:x -----> unimg':S:x
```

So the asymptotical time complexity behaviour of this algorithm is the same as the asymptotical time complexity behaviour of the algorithm defined for "lem:(RS)", because the algorithms defined for the unimg:W:x and unimg':W:x have the same asymptotical time complexity behaviour if the relation W is an extensionally represented relation; in the same way, the algorithms defined for the lem:W and rim:W have the same asymptotical time complexity behaviour when the relation "W" is an extensionally represented relation.

This algorithm is similar to the algorithm defined for the (primitive) "Image" operation. We have to make the substitution below in the algorithm for the "Image" operation in order to define the algorithm for this operation,

unimg ------> unimg'

So the asymptotical time complexity behaviour of this algorithm is the same as the asymptotical time complexity behaviour of the algorithm defined for the (primitive) "Image" operation.

unimg':R:x (Where R is an Extensionally Represented Relation)

This operation, given an individual in the codomain of a relation, finds the set of individuals that are in relation with this individual in the domain of the relation in question. Of course in our case the individuals of the resulting set will be returned to the caller (function "Force" itself) one at a time. We did not define the algorithm for this operation in the extensional representations analysis, so we will define this algorithm here. The algorithm is as follows:

1. Find the relation's record in the relation table, follow the pointer found in the PFRM field of this record, and find the record of the first right member of this relation in the RHT.

2. Hash into the LHT with the argument individual under the relation identifier; find the record of this individual.

3. Reference the incidence vector of the relation with the indices of the records found; if a 1 is found in the corresponding location do step 3-a else to step 3-b.

 a. Take the next integer from the global count, set a pointer to the next record in the RIM set of the relation, hash into the MHASH table with this integer as the identifier and save this pointer in the MHASH table by establishing a record.

 b. Proceed in the RIM set by following the pointer found in the TASE link field of the current RHT record and repeat steps 2 and 3 for the next individual in the RIM set of the relation R.

4. As the unimg':R:x is forced, get the same integer obtained in step 2, by going through the same counting mechanism; hash into the MHASH table and find the pointer save; follow this pointer and find the individuals record in the RHT, then repeat steps 2 and 3.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

If we force the above algorithm repeatedly until no more individuals remain to be returned, we trace through the entire RIM set of the relation R, so we make a number of memory references proportional to "n", where "n" is the

157

cardinality of the RIM set of the relation in question. So the algorithm has the worst case asymptotical time complexity behaviour of O(n).,

(R#S)c!:C

We know that the converse of relation (R#S) is not necessarily an injective function, even though the relation (R#S) is by definition an injective function. So the algorithm is more costly then the algorithm defined for (R#S)!:C. The algorithm is as follows:

1. Force the argument set C, repeatedly.

2. For each individual (which is necessarily a pair) obtained in this manner, take the left individual of this pair, and call it "x". Force unimg':R:x repeatedly, establishing the record of each individual returned, in the LHT with the integer taken from the global count as the identifier. In the same manner take the right individual of the pair and call it "y". Force unimg';S;y repeatedly, establishing the records of the individuals returned in the LHT with the next integer taken from the global count being the relation identifier. In the same manner take the right individual of the pair and call it "y". Force unimg':S:y repeatedly, establishing the records of the individuals returned in the LHT with the next integer taken from the global count being the relation identifier.

3. After the set C is exhausted, take the intersection of the two sets created in the LHT by using the algorithm defined for the "Set intersection" in the extensional representations analysis, and establish the resulting set in the LHT with the next integer taken from the global count as the identifier. Return the first individual of this set, take the next integer from the global count and save the pointer to the next individual's record in the MHASH table with this integer as the identifier.

4. In the subsequent force operations do not perform steps 1 through 3, but perform the counting operation done in each step; i.e., take the integers from the global count and hash with the last integer found in step 3 into the MHASH table. If a pointer is found to be in the MHASH table with this integer as the identifier, return the individual whose record is pointed by this pointer, advance the pointer to the next record of the set obtained in step 3, and save the new pointer in the MHASH table with the same integer being the identifier.

So if we force the above algorithm for the first time, it constructs the set of all individuals to be returned in the subsequent force operations also, and returns the first individual. In the subsequent force operations, steps 1 through 3 are not executed except in order to find the same integer found in step 3. We have to go through the same

counting process on the global count each time the algorithm
is forced.

Now we will do the worst case asymptotically time
complexity analysis of this algorithm.

For each individual of the set C we force the unimg':R
and unimg':S.   Assuming R, S and C are extensionally
represented, we make the number of memory references
proportional to the "n" for each individual of the set C.   If
we assume that the sets C, rim:R and rim:S all have the same
cardinality "n", we make a number of memory references
proportional to the square of "n". Taking the intersection
of two sets (in step 3) costs us a number of memory
references proportional to the "n" (in the worst cast when
the cardinality of those sets are exactly equal to "n").
Because this factor is added to the previous term of the
complexity function, it does not affect the asymptotical time
complexity behaviour of the algorithm.   So we conclude that
the algorithm has the worst case asymptotical time complexity
behaviour of $O(n^2)$.

unimg:(R#S)c = unimg':(R#S)

This algorithm does the same things as the algorithm for
(R#S)!:C does except in this case the argument is not a set
(C); instead it is an individual. So if the algorithm for

160

(R#S)!:C executes on a singleton set, it does the job of this algorithm, so there is no need to rewrite this algorithm here.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

Because the algorithm is the special case of the algorithm for (R#S)!:C, it makes the number of memory references proportional to the "n" for the given argument individual, and it performs the set intersection operation in O(n) time. So with the same assumptions we had in the time complexity analysis of the algorithm for (R#S)!:C, we conclude that the worst case asymptotical time complexity behaviour of this algorithm is O(n).

(R||S)c!:C

This algorithm is similar to the algorithm for (R#S)c!:C except the third step is more costly than the third step of the algorithm of the (R#S)c!:C. The algorithm is as follows:

1. Do steps 1 and 2 of the algorithm defined for the (R#S)c!:C.

2. Start from the beginning of the set obtained from repeated application of unimg':R operation. For each individual found by proceeding in that set, start from the beginning of the set obtained from the repeated application of unimg':S operation and proceed in that set record by

record, by looking up the individual being represented by each record encountered.

3. For each individual pair obtained in step 2, construct a pair relation (as it was done in the algorithm for "Parallel application" in the extensional representations analysis). Take the next integer from the global count, establish the record of the individual (pair) in the LHT with the relation identifier being this integer, and link the records of that kind by their TASE links as they are created.

4. Repeat step 4 of the algorithm defined for the (R#S)!:C.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As we can see the algorithm differs from the algorithm for (R#S)!:C in the third step only. So instead of intersecting the sets, we construct a set which has pairs resulted from pairing up the individuals of those sets. In constructing pairs out of the individuals of the sets obtained in step 1 of this algorithm, we make a number of memory references proportional to the square of "n" where "n" is assumed to be the common cardinality of those sets and the cardinality of the rim:R and rim:S. Because this term will replace the linear term corresponding to the set intersection in the time complexity function of the algorithm of the (R#S)c!:C, and because we already have an order two term in

that complexity function, this additional second degree term will not change the asymptotical time complexity behaviour of the previous complexity function. So we conclude that the algorithm has the worst case asymptotical time complexity behaviour of $O(n^2)$. In practical sense it is more expensive than the algorithm for the (R#S)c!:C.

unimg:(R||S)c:x = unimg':(R||S):x

This algorithm is similar to the algorithm defined for the (R||S)c!:C except the argument is an individual rather than a set. Hence, if we force the (R||S)c!:C only once, this would be equivalent to forcing the unimg':(R||S)c:x, where "x" is the first individual of the set "C". Because of this, we do not need to rewrite this algorithm again.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

Even though we can find the intermediate sets in step 1 of the algorithm referenced above by making a number of memory references proportional to "n" (where "n" is the same as the "n" defined in the time complexity analysis of the algorithm for the (R||S)!:C), we have to pair up the individuals of those sets, which requires a number of memory references proportional to the square of "n". So the worst case asymptotical time complexity behaviour of this algorithm becomes $O(n^2)$.

163

(non:R)!:C, unimg:(non:R):x   (Where R is Represented Exten-
sionally)

When the relation R is represented extensionally we can
do these operations (production of individuals) in a less
complex manner than the algorithms that we will define for
general case.

The algorithms for these operations are exactly the same
as the algorithms defined for R!:C and unimg:R:x that work on
the extensionally represented R, except we will design these
algorithms so that it will accept every 0 found in the hash
incidence vector as 1 and every 1 as 0, by complementing
every entry of the incidence vector tested without changing
the original entry during the operations. By doing that we
do not have to complement all the entries of the incidence
vector by executing the "Complement" algorithm defined in the
extensional representations analysis; instead, our algorithms
that work on the complement of the relation in question,
assume every 0 as 1 and every 1 as 0. Because we are using
essentially the same algorithms defined for R!:C and
unimg:R:x, it is obvious that these algorithms will also have
the same asymptotical time complexity behaviours.

(non:R)!:C (General Case)

This composite set presents some difficulties in defining
the algorithm for it because the resulting set strictly
depends on the tuples of the composite relation R which are

not being represented explicitly. The algorithm is as follows:

1. Force the lem:R repeatedly; take the next integer from the global count; establish the record of each individual obtained in the LHT with this integer as the identifier.

2. Force the set C repeatedly; for each individual "x" obtained in this manner do step 3.

3. Force unimg:R:x repeatedly, if it returns at least one individual increment, the count called "CARD"; for each individual obtained by repeatedly forcing the unimg:R:x, hash into the LHT with the integer taken from the global count (in step 1) as the identifier. Find the record of this individual and increment the integer in the index field of this record (which is not being used in this case).

4. After no more individuals of C remain to be produced, begin from the beginning of the set constructed in the LHT and proceed in that set record by record. For each record found look up the index field; if the integer in the index field is equal to the last integer saved in the counter CARD, delete this record of the individual from the set.

5. Start from the beginning of the set resulting from the execution of step 4, and return the first individual to the caller.

165

6. If the algorithm is forced subsequently, skip steps 1 through 4, but allocate the integer taken from the global count in step 1, and return the next individual that remains to be returned in the set resulting from the execution of step 4 during the first forcing operation.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

Suppose the argument C is an extensionally represented set, and the unimg operation on the composite relation R has an O(n) time algorithm, where "n" is the cardinality of LEM set of the composite relation R. Because we are forcing the unimg:R operation repeatedly with the arguments being each individual of the set C, assuming the set C has the cardinality "n", we make the number of memory references proportional to the square of "n". So under these assumptions, the algorithm has the asymptotical time complexity behaviour of $O(m^2)$. In finding the asymptotical time complexity behaviour of this algorithm, we did not take into account the establishment of the lem:R in the LHT, because the term corresponding to this operation is added to the term which we found above, and in most of the cases this term is a linear term, so it does not affect the asymptotical time complexity behaviour of the algorithm. Note that this cost should be attributed to the first forcing operation done on this composite set; the subsequent force operations cost

us constant time, because we established the set of individuals to be returned when this composite set was forced the first time. That means the subsequent force operations effectively force an extensionally represented set and each force operation becomes a constant time operation.

fan:R!:C

Producing the individuals of this set is a costly operation which requires repeated execution of the expensive "Image" operation. The algorithm is as follows:

1. Take the first integer from the global count; force the lem:R repeatedly and increment COUNT 1 for each individual produced; establish each individual produced in the LHT, with this integer as the identifier. In the same manner, force the rim:R repeatedly, hash into the LHT with each individual under the above relation identifier. If this individual does not have any record in this set, increment COUNT 2, else do nothing.

2. Take the next integer from the global count; force the composite set R!:C repeatedly; for each individual of the set C obtained during that operation, hash into the LHT with the next integer taken from the global count as the identifier and establish the record of this individual. Call the set resulting from this operation C'. In the same manner take the next integer from the global count and establish the record of each individual of the set resulting from

167

repeatedly forcing the R!:C with this integer as the relation identifier.

3. Force the Image operation with the set (other than the C') resulting from the execution of step 2, being the argument; establish the records of the resulting set's individuals in the LHT under the same relation identifier obtained in step 2, and in the same manner as explained in step 2.

4. Proceed in the same manner by every time taking the set resulting from the previous step and forcing the "Image" operation repeatedly on this set in order to obtain the next set and each time increment a counter, namely "M". Increment the counter COUNT 3 for each record of the resulting set created.

5. Do step 4 until M = COUNT 1 + COUNT 2 or COUNT 3 = COUNT 1.

6. Start from the beginning of the set C', for each individual found by proceeding in this record by record, hash into the LHT with the resulting set's identifier, (i.e., integer obtained in step 2) and if this individual does not have a record in this set, establish the record of this individual in that set.

7. Return the set C' to the storage pool.

8. Return the set constructed in step 1 to the storage pool.

168

9. Return the first individual of the resulting set. Delete the record of this individual from that set.

10. If the algorithm is forced subsequently, skip step 1 through 8 and execute step 9, but under any condition take the integers from the global count which are obtained in step 2.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We defined the above algorithm for general case, in which the relation R is a composite relation. But because it is not possible to list down an infinite number of composite relations, we will analyze the case in which the relation R is an extensionally represented relation.

In the worst case each of the intermediate sets obtained in step 4 has the cardinality "n-1", where "n" is the cardinality of the LEM set of the relation R. It is not equal to "n" because this is the stopping condition of the algorithm. This does not make any difference in the asymptotical time complexity behaviour of the "Image" algorithm, so we make a number of memory references proportional to the square of "n" for each repetition of step 4. We know that in the worst case step 4 executes M-2 times where M is the cardinality of the MEM set of the relation. Let's assume the LEM and the RIM set of the relation are disjoint and M=2*n, where "n" is again assumed to be the

169

common cardinality of the LEM set and the RIM set of the relation. So it is clear that we are making a number of memory references proportional to the square of "n", 2*n times. As a result we can write the leading term of the complexity function as follows:

K*(n^3)

where K >= 2

So we conclude that under these conditions the algorithm has the worst case asymptotical time complexity behaviour of O(n^3). Note that we did not take into account the cost of execution of steps 1, 6, 7 and 8 because the terms of the complexity function associated with those steps would be linear terms and would not affect the asymptotical time complexity behaviour of the algorithm.

(san:R)!:C

This algorithm is similar to the algorithm defined for the (fan:R)!:C, except we do not create the set C', and we have to omit steps 6, 7 and 8. So the asymptotical time complexity behaviour of this algorithm is the same as the asymptotical time complexity behaviour of the algorithm defined for (fan:R)!:C, because the asymptotical time complexity behaviour of steps 6, 7 and 8 did not have any affect on the asymptotical time complexity behaviour of the algorithm defined for the (fan:R)!:C.

As we mentioned earlier, we will generalize the set operations: Intersection, Union, and Difference in the context of producing the individuals of the composite sets. That means these operations will have special meaning in our function "Force". Each of those set operations, being binary operators, may take on any kind of composite sets as operands, and produce the individuals of the resulting composite set. The algorithms that we will define will establish temporary sets in the LHT in order to produce the individuals of the resulting sets efficiently, like was done in some of the algorithms above. There exists another technique which prevents us from constructing these temporary sets, and requires doing membership tests for each individual to be produced. We will give an example of this technique in defining the algorithm for the "Intersection" operation, and explain the reasons why this kind of algorithms is costly. On the other hand, our technique uses more storage, but we reuse that storage many times by maintaining a storage pool as explained before. Our technique is aimed at splitting the terms of the time complexity function rather than nesting the terms in each other by increasing the exponent of the terms. This can be done simply by producing and saving some sets, temporarily in advance, then testing the individuals to be produced against those sets, rather than every time producing the sets which the individual being produced is to be tested

171

against. Because those temporary sets will be established in the LHT, the membership test operations will be constant time operations.

Set Intersection in Producing the Composite Set Individuals (and):

In this algorithm we will first construct one of the composite operand sets explicitly in the LHT, then we will force the other operand set as the composite set constructed with "and" operation, is forced. Each time this set is forced we test the individual to be produced against the set constructed in the LHT; if the same individual also exists in that set, the individual in question is produced, otherwise we continue to force the other operand composite set until we find an eligible individual to produce. The algorithm is as follows:

1. Take the next integer from the global count and force the right operand set repeatedly. For each individual found in this manner, hash into the LHT with the integer taken from the global count above as the relation identifier. Establish the record of this individual in the LHT if it does not have any record under this identifier already.

2. Force the left operand set, get the individual returned, and hash with that individual into the LHT with the integer obtained in step 1 as the relation identifier. If there exists a record for this individual in that set, return

172

this individual as the result, otherwise force the left operand set again in order to continue with the next individual of that set.

3. If none of the individuals of the left operand set is found to be in the set (which is established in the LHT) return "nil".

4. If the main composite set is forced subsequently skip step 1, but under any condition take the integer obtained in step 1 from the global count.

As can be seen the above algorithm does not care about the kind of composite operand sets because the "Force" operation being performed on the composite operand sets is defined for all kinds of composite sets and that is what we mean by generaliztion.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As we have seen in the previous algorithms, most of the operations eventually result in the execution of the "Unit image" operation on the extensionally represented relations, and we know that the "Unit image" (algorithm, when it works on the extensional relations, has the worst case asymptotical time complexity behaviour of $O(n)$. Except some special cases for which we defined the algorithms in this section, we expect most of the exhaustive production operations to be linear time operations, because step 1 and step 2 of this

algorithm are independent steps. If the execution of step 1 and the execution of step 2 have linear time complexity behaviour. clearly the algorithm will have the linear time complexity behaviour. On the other hand, if any one of the steps 1 or 2 or both have the order two time complexity behaviour, clearly the algorithm will have the order two time complexity behaviour, and so on. Hence we conclude that the worst case asymptotical time complexity behaviour of the exhaustive execution of the "Force" operation on the composite operand sets, and most of the time we expect the worst case asymptotical time complexity behaviour of this algorithm to be either $O(n)$ or $O(n^2)$.

Now suppose we defined the above algorithm by using the other technique we mentioned above, in which case we would force the left operand set and obtain the individual, then we would force the right operand set repeatedly and compare the individual in question with each individual produced. By doing that we would recover from the explicit construction of the right operand set, but each time we force the main composite set we would produce all the individuals of the right operand set which is very inefficient in time. So our algorithm saves the individuals of the left operand set temporarily and subsequent "Force" operations become effectively the same as forcing the right operand set.

174

Difference Operation (-)

This algorithm is almost the same as the algorithm defined for the intersection operation. In order to make the distinction clear, we will rewrite step 2 of the algorithm defined for the intersection operation:

2.   Force the left operand set, get the individual returned, and hash with this individual into the LHT with the integer obtained in step 1 being the identifier. If there exists a record of this individual in that set, force the left operand set again in order to continue with the next individual of the left operand set, otherwise return this individual.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We can use the same argument as we have used in the time complexity analysis of the algorithm for the "Intersection" operation, and we can say the same things about the worst case asymptotical time complexity behaviour of this algorithm.

Union Operation (or)

We could define this algorithm simply as follows:

1.   Force the left operand set, get the individual and return it, if there remains any individual to be returned in the left operand set.

2.   If no individuals of the left operand set remain to

be returned, force the right operand set, get the individual and return it, if there remains any individual to be returned in the right operand set.

3. Do the above steps as the main composite set is forced.

The above algorithm may produce the same individuals repeatedly if the operand sets are not disjoint. So we will again save the individuals of the left operand set while we are producing these individuals in order to remember which individuals were produced before and not produce them again. The algorithm is as follows:

1. Take the next integer from the global count, call this integer "I". Force the left operand set, get the individual returned, hash into the LHT with this individual and, with the integer "I" as the relation identifier, establish the record of this individual in the LHT under the relation identifier "I". Return the individual in question.

2. Repeat step 1 as the main composite set is forced. Construct a set structure in the LHT out of the records of the individuals produced, while these individuals are being produced.

3. Do step 2 until no more individuals of the left operand set remain to be produced , as the main composite set is forced.

4. If no individual of the left operand set remains to be produced and if the main composite set is forced subsequently, begin producing the individuals of the right operand set one at a time as the main composite set is forced.

5. For each individual produced in the manner explained in step 4, hash into the LHT with this individual and with the integer I as the relation identifier. Check if this individual has a record in that set; if so do step 5-a else do step 5-b.

a. Force the right operand set in order to continue with the next individual of the right operand set, and go to step 5.

b. Return the individual in question.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As can be seen the algorithm has essentially the same structure as the algorithms defined for the "Intersection" and "Difference" operations and we can use the same kind of argument in this case also; i.e., the asymptotical time complexity behaviour of this algorithm depends on the asymptotical time complexity behaviour of the execution of the "Force" operation on the operand (composite) relations.

177

D. MEMBERSHIP TEST ALGORITHMS

As we mentioned earlier, in some operations we want to test the membership of an individual in a given set. We may do this in two ways:

1. We produce the individuals of the composite set one at a time and compare each individual produced with the individual in question.

2. We may define a less costly algorithm for each kind of composite set which focuses on the individual and does less work in testing the membership of this individual in the given composite set.

The first method is a costly sequential method since, in some cases, we may produce all the individuals of the composite set. The second method defines algorithms for membership tests on each kind of composite set that we have given the algorithm for in chapter C, whenever the cost is less than the cost of using the method 1. On the other hand Method 2 uses up some memory which can be reused by maintaining a storage pool.

In defining the algorithms for the membership test on the different kinds of composite sets we will use a new notation which short cuts a lot of detailed description that the reader got used to while reading the previous sections. New notation:

```
Fr(composite set)    = Force the composite set repeatedly.
---set--->           = Create a set out of the individuals
                       produced by the operation on the left
                       of the arrow, in the RHT by
                       attributing a unique identifier to
                       this set, which is represented as a
                       capital letter on the right of the
                       arrow.
---tx--->            = Transmit the resulting set/individual
                       to the next operation as the argument
                       of that operation.
--test-each-in-->    = Do the membership test for each
                       individual produced by forcing the
                       composite set shown on the left of
                       the arrow, to see if it is in the set
                       which is shown on the right of the
                       arrow.
---any--->           = If any individual is found to be a
                       member of the set indicated on the
                       left of the arrow, output "true".
while C2;            = While producing the individuals of
                       the set C2, do the step indicated on
                       the right of the ";" also.
```

```
---tx--->varA         = Assign the logical value (true/false)
                        indicated on the left of the arrow,
                        to the boolean variable "A".

---is-in-->           = If the individual indicated on the
                        left of the arrow is in the set
                        indicated on the right of the arrow,
                        output "true".

true:{statement}      = If the input condition was "true"
                        then the statement indicated in the
                        braces is true.

false:{statement}     = If the input condition was "false"
                        then the statement indicated in the
                        braces is true.

isempty:C             = If the given set C was the empty set,
                        output "true" otherwise output
                        "false".

-                     = Set difference operation.

and                   = Set intersection operation/logical
                        "and" operation.

or                    = Set union operation/logical "or"
                        operation.

z                     = The individual to be tested for
                        membership.

x                     = Argument individual.

C                     = Argument set.
```

180

```
left(x)                 = Left individual of the pair "x".
right(x)                = Right individual of the pair "x".
```

In these algorithms extensionally represented temporary sets will be given unique identifiers from a global count which is different than the global counts associated with the high level composite sets in the function "Force" case. These temporary sets will be established in the RHT instead of LHT in order to prevent the possible collisions that may occur because of the sets created by the function "Force" since they both use integers as identifiers.

In order to make the notation clear, we will define two algorithms in a way as we have done before, and we will explain the correspondence between the notation and the steps of those algorithms. The algorithms referring to the remaining operations can be found in Appendix D, which are expressed by using the notation given above.

(R-S)!:C

Given an individual to be tested for membership in the composite set (R-S)!:C, we have to find out if a tuple of R exists which has this individual as the left individual. The second condition is the right individual of this tuple must be in the argument set C and the last condition is this tuple must not exist in the relation S. The algorithm which checks those conditions is given below:

181

1. Take the next integer from the global count. (Note that this global count is not the same as the global count used in the function "Force"). Force the unimg':R:z repeatedly and for each individual obtained do step 2. (* In our notation this step can be expressed as: Fr(unimg':R:z) *)

2. Hash to the RHT with the individual obtained and with the integer taken from the global count as the identifier, establish the record of this individual in the RHT.

3. Link the records of the individuals to each other by their TASE links as they are created. (* Assuming the integer identifying the set produced in the RHT represented as C1, in our notation, all of the above steps can be expressed as follows:

Fr(unimg':R:z) ----set----> C1   *)

4. Force repeatedly unimg':S:z; for each individual obtained in this manner, hash into the RHT with this individual, with the integer taken from the global count in step 1 being the relation identifier. (In this case the relation identifier is used to identify the set established in the RHT). Check if this individual has a record in that set; if so delete this record else do nothing and continue with the next individual of the unimg':S:z. (* In our notation this step can be expressed as follows:

Fr(unimg':S:z) ---set---> C2'

while C1'; C1 - C2   ----set----> D'

where C2' is a place holder, because we are not constructing that set explicitly. The second statement expresses that, while producing the individuals of the set C2', get the set difference (C1-C2') also and call the resulting set D'. Note that the identifier of the resulting set (D') is actually the integer that we took from the global count in step 1 which is represented as C1, but we used D' as identifier in order to emphasize the fact that the set C1 may change after that operation. *)

5.   Force the argument set C repeatedly. Take each individual returned and check if this individual is in the set resulting from the execution of step 3 by hashing into the RHT with this individual under the relation identifier (integer) obtained in step 1. If so, conclude that the individual "z" is in the set (R-S)!:C and quit forcing the argument set C. Otherwise continue to force the set C in order to test the remaining individuals of the set C in the manner explained above. (* In our notation, this step can be expressed as :

   Fr(C)--test-each-in--> D'--any-->true{z is in the set}

So, in our notation, the complete algorithm can be written as follows:

   Fr(unimg':R:z) ---set---> C1

   Fr(unimg':S:z) ---set---> C2'

183

while C2'; Cl - C2' ---set---> D'

Fr(C)--test-each-in--> D'--any-->true{z is in the set} *)

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We can write the time complexity function of this algorithm as follows:

$f = K + L + M$

where:

K corresponds to the steps 1, 2 and 3.

L corresponds to step 4.

M corresponds to step 5 of the algorithm.

and K, L and M are not constants as opposed to our convention. Instead each of the K,L and M represents a term of the complexity function. The worst case asymptotical time complexity behaviour of the algorithm is exactly the same as the cost of the one of the terms K, L or M indicated above; i.e., if the most of exhaustively forcing the set C, (in step 5) has the worst case asymptotical time complexity behaviour of $O(n^2)$ and the other terms have linear behavior, the worst case asymptotical time complexity behaviour of this algorithm becomes $O(n^2)$, etc.

As we can see, the cost of the algorithm strictly depends on the type of composite sets we are forcing in the various steps of the algorithm. In general we can say that the composite sets that are made up of the "unimg" operation have

184

linear algorithms in the production of the individuals. Assuming the relations R, S and the set C are extensionally represented we conclude that the worst case asymptotical time complexity of this algorithm is $O(n)$, because we know that the algorithms for the production of the individuals of the composite sets, unimg':R:z, unimg':S:z and the extensionally represented set C, have linear behaviour and an algorithm which embeds the sequential execution of those algorithms will also have linear behaviour.

unimg:(RS):x :

The algorithm can be defined as follows:

1. Take the next integer from the global count. Force the composite set, unimg:S:x repeatedly; for each individual obtained in this manner hash into the RHT with this individual and with the integer taken from the global count as the relation identifier (call this identifier Cl). Establish the record of this individual in the RHT. Link the records of the individuals created in the RHT by their TASE links as they are created. (* In our notation, this step can be expressed as follows:

    Fr(unimg:S:x) ----set----> Cl   *)

2. Force the composite set, unimg':R:z repeatedly; for each individual obtained in this manner hash into the RHT with this individual and with the relation identifier Cl. If this individual has a record in the set constructed in step

1, conclude that the individual being tested for membership is in the set, otherwise continue with the next individual of composite set unimg':R:z by forcing it again. (* In our notation, this step can be expressed as follows:

Fr(unimg':R:z) ----set----> C2'

while C2': Cl and C2' ----set----> D'

while D'; isempty(D') ----tx----> false{z is in the set}

where C2' is again a place holder set identifier, because we are not constructing the extensional representation structure for this set; instead we are producing its individuals. The second statement means: "while producing the individuals of the set C2', try to get the intersection of the sets Cl and C2', and call the resulting set D'", which we will not construct the extensional representation for. The last statement means as soon as an individual of the non-existing set D' is found, conclude that the individual being tested for membership is in the set. As can be seen, we are using the identifiers of some intermediate sets even though we are not representing them extensionally in order to make the algorithms as understandable as possible. *)

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We can repeat the same argument as we have done above, as follows: It is clear that the algorithm has the worst case asymptotical time complexity behaviour of O(n) if the

186

algorithms for exhaustively forcing unimg:S:x and unimg':R:z
have linear time complexity behaviour. So if we assume that
the relation R and relation S are extensionally represented
relations we conclude that the algorithm has the worst case
asymptotical time complexity behaviour of O(n), because we
know that the algorithms for forcing the unimg:S: and the
unimg':R: have the linear time complexity behaviour in the
case the R and S are extensionally represented relations.

We will not do the complexity analysis for the remaining
algorithms, because they are all similar to each other and
present the same time complexity characteristics as the above
examples by having disjoint steps and using the (unimg:)
and/or (unimg':) operations.

E. FUNCTION APPLICATION ALGORITHMS

As we have done in the other operations before, we will
define an algorithm for each kind of composite relation which
can be applied to an individual. In chapter 3 we listed down
the kinds of composite relations and mentioned that, we could
define an arbitrary number of different composite relations
by substituting the composite relations in each other as the
operand relations. Because the function application
operation will be defined for each kind of composite
relations in terms of the unimg, unimg' and function
application operations on the operand relations, and because
the unimg, unimg', and function application operations are

187

defined for each kind of composite relation, no confusion arises. The reader should think of the operand relations R and/or S in each kind of composite relation as another composite relation or an extensionally represented relation. In the time complexity analysis we will assume the operand relation/relations, R and/or S as extensionally represented relation/relations, because we are unable to do a complexity analysis on the infinite number of composite relations that may be obtained by substituting the other composite relations in the operand relations/relation. In these algorithms the temporary sets are given integer identifiers from the same global count we used in the membership algorithms, because both kinds of algorithms are in the main body of the interpreter and these sets are established again in the RHT. The algorithms are given below:

(R&S):x

The algorithm for this composite relation can be defined as follows:

1. Take the next integer from the global count and force the composite set unimg:R:x; take each individual returned and hash into the RHT with this individual, and with the relation identifier being the integer taken from the global count. Establish the record of this individual in the RHT. Link the records of the individuals to each other by their TASE links as they are created.

188

2.   Force the set unimg:S:x repeatedly; take each individual returned and hash into the RHT with this individual and with the integer obtained in step 1 as the relation identifier.   If this individual has a record in that set, return this individual, quit forcing unimg:S:x and return the set constructed in the RHT to the storage pool. Otherwise continue with the next individual of the unimg:S:x.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

In the worst case the sets unimg:R:x and unimg:S:x may be disjoint.   In that case we force the composite set unimg:S:x exhaustively.   By assuming the relations R and S are extensionally represented relations, in which case the exhaustive force operations on the composite sets unimg:F:x and unimg:S:x have linear behaviour, we conclude that the worst case asymptotical time complexity behaviour of the algorithm is $O(n)$ where "n" is assumed to be the common cardinality of the sets lem:R and lem:S.   The algorithm has a linear asymptotical time complexity behaviour because it incorporates the sequential execution of two linear algorithms.   Note that we have another linear term which corresponds to disconnecting of the set created in step 1 from the RHT and returning it to the storage pool.   This operation is explained many times in the algorithms for the extensional representation techniques and is shown to have

linear behaviour, so adding this term to the complexity function of the above algorithm would not change the asymptotical time complexity behaviour of the algorithm.

(R|S):x

Among all the algorithms that we will define, the simplest one is the algorithm for the union operation. The algorithm is as follows:

1. Apply relation "R" to the argument individual "x"; if the individual is found, return that individual otherwise do step 2.

2. Apply relation "S" to the argument individual "x"; if an individual is returned, return this individual otherwise call error routine.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We know that the algorithm for the function application operation on the extensionally represented relations has a constant time, time complexity behaviour, because step 1 and step 2 are disjoint steps. By considering the worst case, in which we can not obtain an individual by executing step 1, we would write the time complexity function of this algorithm as follows:

$$f = K+K = 2*K$$

where K is the constant number of memory references made by the function application algorithm defined in the extensional

representation analysis. So we conclude that the algorithm

has the worst case asymptotical time complexity behaviour of

$O(c)$.

(R-S):x

This algorithm is very similar to the algorithm that we

defined for the composite relation (R&S), but we have to

modify step 2 of that algorithm slightly. So we rewrite step

2 as below:

2.   Force the set unimg:S:x repeatedly; take each

individual returned, hash into the RHT with this individual

and with the integer obtained in step 1 as the relation

identifier. If this individual has a record in that set,

delete this record. If after the set unimg:S:x is exhausted,

the set in the RHT still has the records of some individuals,

return the individual represented by the first record of that

set. Return the resulting set to the storage pool.

Now we will do the worst case asymptotical time

complexity analysis of this algorithm.

The asymptotical time complexity behaviour of the

algorithm is essentially the same as the asymptotical time

complexity behaviour of the algorithm defined for the

composite relation (R&S), but the difference is: in step 2,

we force the composite set unimg:s:x exhaustively under any

conditions. So the average case time complexity behaviours

of these algorithms differs but they have the same worst case
asymptotical time complexity behaviour.

(non:R):x

   We can define this algorithm in terms of the previous
algorithms we defined. The algorithm is as follows:

   1. Force the composite set:

   (lem:R - (unimg:R:x))

once, take the resulting individual and return it.

   Now we will do the worst case asymptotical time
complexity analysis of this algorithm.

   We intend to do this analysis so that we can also show
the reader how to find the time complexity behaviour of a
segment of a relational program.

   Assume that the relation R is represented extensionally.
According to the definition of "-" (difference) operation in
the function "Force", we force the composite set unimg:R:x
repeatedly, and construct a set in the LHT out of the
individuals returned, because we assumed that the relation R
is represented extensionally. In order to construct this
set, we make a number of memory references proportional to
"n", where "n" is the cardinality of the LEM set of the
relation. Then we force the composite set lem-R, and we test
if the individual returned is in that set; if so we return
this individual; otherwise we continue to do the same thing

192

for the next individual of the lem:R by forcing the lem:R
again. Because in the worst case:

lem:R = unimg:R:x

We force the lem:R until no more individuals remain to be
produced, and we make a number of memory references
proportional to "n", where "n" is again the cardinality of
the LEM set of the relation R. Because those two exhaustive
sequences of "Force" operations are made one after
another (i.e., the steps are disjoint), the terms of the
complexity function associated with those steps should be
added rather than multiplied, so the resulting complexity
function will have linear behaviour. Under those
considerations, we conclude that the worst case asymptotical
time complexity behaviour of the algorithm is O(n).

Note that we do not need this complex algorithm if the
relation R is extensionally represented because, like we have
done in the unimg and image operations on the complement of a
relation, we can define an algorithm which assumes the 1's of
the incidence vector as 0's and vice versa. We will define
that algorithm next, but we want to emphasize again that the
above algorithm refers to the general case where R can be any
composite relation, but we have to assume R as an
extensionally represented relation in order to be able to do
the time complexity analysis.

(non:R):x (Where R is Represented Extensionally)

This algorithm is different from the algorithm defined for R:x in the extensional representations analysis. It can be defined as follows:

1. Hash to the RHT with the argument individual under the given relation identifier; find the record of this individual.

2. Start from the beginning of the RIM set of this relation and proceed in this set record by record by following the TASE links between the records.

3. For each pair of individuals (tuple) found in steps 1 and 2, reference the incidence vector and return the left individual of the first tuple for which a 0 is found in the corresponding incidence vector location.

Now we do the worst case asymptotical time complexity analysis of this algorithm.

In the worst case all the left members of the relation may be in relation with the argument individual which is in the RIM set of the relation. In that case we trace exhaustively the LEM set of the relation and make a number of memory references proportional to "n", where "n" is the cardinality of the LEM set of the relation. So the algorithm has the worst case asymptotical time complexity behaviour of $O(n)$.

194

$(R||S):x$

The algorithm for this composite relation can be defined as follows:

1.   Extract the left individual of the argument individual "x" (which is necessarily a pair), apply relation R to this individual, and save the individual returned.

2.   Extract the right individual of the argument individual "x", apply relation S to this individual, and save the individual returned.

3.   Construct a pair relation out of the individuals saved in step 1 and step 2, and return the pointer to the record of this pair (which is established in the relation table) to the caller.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We know that the algorithm for the function application on an extensionally represented relation has the asymptotical time complexity behaviour of $O(c)$, and we invoke this algorithm two times in steps 1 and 2.   In addition we construct the pair relation out of the individuals resulting from those function applications in constant time.   Because the steps 1, 2 and 3 are disjoint steps, the terms of the complexity function associated with those steps are added, and the worst case asymptotical time complexity behaviour of this algorithm automatically becomes $O(c)$.

(R#S):x

The algorithm for this composite relation is similar to the algorithm for the composite relation (R||S), except, we apply relations R and S directly to the argument individual "x" in steps 1 and 2; obviously, it has the same asymptotical time complexity behaviour as the algorithm for comnposite relation (R||S).

(RS):x

The algorithm for this composite relation can be defined as follows:

1. Apply the relation S to the argument individual "x"; take the individual returned and call it "y".

2. Apply the relation R to the individual "y"; take the individual returned and return it.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We know that the function application costs us constant time if the relation in question is represented extensionally, so the steps 1 and 2 of the above algorithm cost us constant time each. Because the steps are disjoint, we conclude that the worst case asymptotical time complexity behaviour of this algorithm is also constant time.

Meta Application ((R::S):x)

We did not define an algorithm for this operation in the extensional representations analysis because it was hard and

196

infeasible to construct the extensional representation structure for this case. Before we go into the reasons for doing that, we will summarize what the operation does.

This operation applies the right operand relation to the argument individual and records the individual obtained, then it applies the left operand relation to the argument individual. If the individual obtained is a relation, it applies this relation to the individual recorded above and returns the resulting individual.

Now suppose we tried to construct the extensional representation structure for this operation. We would need to apply each of the relations known by the system so far to all individuals known by the system, and we would construct the RIM set of this relation out of the individuals resulting from those application operations. Then we would need to apply each of the relations known by the system to all of the individuals of the RIM set of the relation and construct the LEM set of this relation out of the individuals resulting from those application operations. As can be easily seen, the process is very costly and the resulting relation should be updated as soon as a new relation and/or a new individual is introduced to the system. On the other hand, if we do this operation intensionally, no problem arises. The algorithm for this operation is given below:

197

1. Apply the right operand relation to the argument individual; record the individual returned.

2. Apply the left operand relation to the argument individual and take the individual returned, hash into the relation table with this individual and check if it is a relation. In the case this relation is not in the relation table but represented intensionally (i.e., if it is a composite relation), hash into the relation table with each relation identifier out of which this composite relation is constructed. In any case, if this relation is found to be applicable to the argument individual in question, apply this relation to the individual recorded in step 1 and return the resulting individual. Otherwise, call the error routine.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

Let the left operand relation (R) and right operand relation (S) be extensionally represented relations. We know that the function application operation on extensionally represented relations has the constant time asymptotical time complexity behaviour. As a result, step 1 and step 2 make a constant number of memory references so we conclude that the worst case asymptotical time complexity behaviour of this algorithm is $O(c)$ in the case the operand relations are extensionally represented relations.

(fan:R):x

This operation is an expensive operation relative to the other function application operations on the various composite relations. The algorithm is as follows:

1. Force the composite set:

lem:R|rim:R

repeatedly, count the number of individuals produced, i.e., obtain the cardinality of the MEM set of the relation, and call this "M".

2. Apply relation R to the argument individual and take the individual obtained, apply the relation R to this individual again and take the individual obtained. Repeat the application operation in the same manner by each time applying the relation R to the individual obtained from the previous application operation M times or until an application operation returns "nil".

3. In the first case when the application operation is repeated M times, return the last individual obtained. In the second case when an application operation returns "nil", return the individual obtained from the previous application operation which has not returned "nil".

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We know that the term of the time complexity function corresponding to step 1 of the algorithm has the linear time

complexity behaviour because in that step we make a number of memory references proportional to M (where M is the cardinality of the MEM set of the relation in question), in the case the relation R is an extensionally represented relation. The term of the complexity function corresponding to step 2 also has linear time complexity behaviour, because each function application operation makes a constant number of memory references and we repeat the function application operation M times in the worst case. So in step 2 we make a number of memory references proportional to. M, where M is the cardinality of the MEM set of the relation in question. Because steps 1 and 2 are disjoint steps we add the terms of the complexity function corresponding to step 1 and step 2 together and obtain a linear time complexity function. So we conclude that the worst case asymptotical time complexity behaviour of this algorithm is O(n) where "n" is the cardinality of the MEM set of the relation in question.

(san:R):x

This operation is not a well defined operation because, given the argument individual, we can check if it is a member of the MEM set of the relation in question and if it is, we return the argument individual itself as the resulting individual. This property of the operation originates from the fact that the second ancestral of a relation is the reflexive transitive closure of the relation. Because the

second ancestral of a relation has to be a reflexive relation, if the given (argument) individual is in the RIM set of the relation, it has to be in the LEM set also. So after doing the above membership test, we can immediately return the argument individual itself as the resulting individual. Thus this operation does not have any meaning from the user's point of view.

Even though it does not have meaning, we will use this operation as a part of another operation, which is equivalent to "while" loop in conventional languages. This operation is explained below.

C/((san:R):x)

This operation can be viewed as a while loop, in which the left restriction operation imposes the condition of the loop and "san:" operation forces the loop to iterate. Since we did not define the "san:" operation we will accept this operation like a completely new operation and we will define an algorithm for it. The algorithm is as follows:

1. Test if the argument individual x is in the MEM set of the relation R.

2. If it is in the MEM set of the relation R, test if it is in the set C; return this individual as the result; otherwise, do step 4.

3. If it is not in the MEM set of the relation then call the error routine.

201

4. Apply relation R to the individual x; take the individual returned and test if it is in the set C. If so return this individual else do step 5 for this individual.

5. Apply relation R to the individual; take the individual returned and test if it is in the set C. If so return this individual else repeat the step for this individual.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As it is in the "while" loop of conventional languages, if the condition is never met the algorithm goes into an infinite loop. As can be seen we perform one function application operation and one membership test operation in each iteration of the loop. So the worst case asymptotical time complexity behaviour of the algorithm is the same as the worst case asymptotical time complexity behaviour of either the membership test operation or the function application operation (depending on which operation is more expensive), times the number of iterations.

If we assume that the set C and the argument relation R are represented extensionally, it is obvious that each iteration costs us constant time because the membership test operation is a constant time operation when the set in question is represented extensionally. In the same way, when the relation is represented extensionally, the function

202

application operation becomes a constant time operation. So the only variable remaining to be taken into account is the number of iterations. We conclude that the algorithm has the worst case asymptotical time complexity behaviour (in the case of the set C and the relation R represented extensionally) of O(n), where n is the number of iterations.

In the same manner we can define an operation which is equivalent to the "repeat" loop in the conventional languages. This operation is explained below.

C/((fan:R):x)

Even though we defined an algorithm for the operation "(fan:R):x", we can not make use of it in this case. We have to accept this operation as a stand alone operation. Because the operation "(fan:R):x" finds an individual to be returned as a result and quits at that point; on the other hand in this operation we want the loop to continue if the resulting individual is not an element of the set C. The algorithm for this operation is the same as the algorithm of the operation "C/((fan:R):x)" except, we do not include the step 1, 2 and 3 of that algorithm. So the same worst case asymptotical time complexity analysis can be done in this case too.

Reduction Operation (@:(R,f)):(i,x)

This operation is aimed at reducing sequences. The operation takes a sequence R, a function f (which takes a pair and returns an individual), an initial value i, and the

203

first individual of the sequence x. It applies the function f to the argument pair, takes the resulting individual, and constructs a new pair in which the left component is the individual obtained above and the right component is the next individual of the sequence R. This new pair goes under the same process as the argument pair did and this process continues until the end of the sequence is encountered. We will give an example to make the operation clear to the reader. Example:

Suppose the sequence R (a relation) is defined as the integers from 1 to 9 (which has the tuples like: <1,2>, <2,3>, <3,4>, .. and so on) and the function f is defined as the addition (+); i.e., it takes a pair, adds up the left and right components and returns the result. Suppose the initial value is given as 0 and the first individual of the sequence is given as 1. (In some cases we may want to begin with another individual of the sequence depending on the application). So the operation to be performed is:

(@:(R,+)):(0,1)

The operation first adds 1 to 0 and looks up the next individual of the sequence which is 2. It constructs the pair (1,2) and applies function f to this pair again. It takes the result (3), looks up the next individual of the sequence which is 3 and constructs the new pair (3,3). It continues in the same manner until it creates the pair

(45,eos), where "eos" represents the end of the sequence. At this point it returns the result (45).

We define the algorithm of this operation as follows:

1. Get the identifiers of the sequence R (a relation) and the function f (a relation).

2. Get the argument pair.

3. Apply the function f to the argument pair; take the individual returned and call it W.

4. Apply the sequence R to the right component of the pair; take the individual returned and call it Z.

5. If the individual obtained in step 4 is the end-of-sequence mark, then do step 7. Else construct a pair in which the right component individual is Z and the left component individual is W.

6. Go to step 3 with the pair constructed in step 5 being the argument pair.

7. Take the left component of the pair and return it.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As can be sen we perform two function appliction operations for each individual of the sequence. If the sequence R and the function f are extensionally represented relations, then the function application operations on those relations become constant time operation. So we make a constant number of memory references for each individual of

the sequence. Under this consideration we conclude that the worst case asymptotical time complexity behaviour of this algorithm is O(n), where n is the number of individuals in the sequence.

Rc:x

In the preprocessing phase many of the converse composite relations such as (R|C)c, (R&S)c, etc. reduce down to the primitive converse relations such as Rc, Sc, where R and S are represented extensionally. Hence we have to define the algorithm for the function application operation which works on the converse of an extensionally represented relation. The algorithm is as follows:

1. Hash to the LHT with the argument individual under the relation identifier in question. Find the record of this individual and follow the pointer found in the PRRM field of this record. Find the RHT record of the individual which is in relation with the argument individual under the relation in question.

2. Return the individual which is represented by the RHT record found in step 1.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

It is obvious from the algorithm that we make a constant number of memory references for doing this operation. So the

206

algorithm has the worst case asymptotical time complexity behaviour of O(c), as it was in the (R:x) case.

Some converse composite operations can not be reduced to the primitive operation given above, and should have specially defined algorithms; those algorithms are given below:

(R||S)c:x

The algorithm for this composite relation is as follows:

1. Apply the relation "Rc" to the left individual of the argument individual "x" (which is necessarily a pair), get the individual returned and save it.

2. Apply the relation "Sc" to the right individual of "x", get the individual returned and save it.

3. Construct a pair relation out of the individuals saved in steps 1 and 2; return the pointer to the record of this pair relation which has been established in the relation table.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

If the relations R and S are extensionally represented relations, we make a constant number of memory references in steps 1 and 2. In addition we know that we construct a pair relation by making the constant number of memory references. So in the above algorithm we make a constant number of memory references in order to obtain an individual as a result.

207

Under the above consideration we conclude that the algorithm
has the worst case asymptotical time complexity behaviour of
O(c).

(R#S)c:x

This algorithm is more expensive than the algorithm for
the (R||S)c:x, because the intersection of the set of
individuals that are in relation with the left individual of
the argument "x" (which is necessarily a pair) under the
relation R, and the set of individuals in relation with the
right individual of the argument "x" under the relation S
should contain at least one individual which is to be
returned as a result. So in this case we have to execute the
"Unit image" operation on the left individual and on the
right individual of the argument individual (pair) "x". The
algorithm is as follows:

1. Force the composite set:

(unimg':R:(left(x))) and (unimg':S:(right (x)))
once and return the resulting individual.

Now we will do the worst case asymptotical time
complexity analysis of this algorithm.

Assume the relation R and the relation S are
extensionally represented relations. Because in the worst
case (when the unimg':R:left(x) and the unimg':S:right(x) are
disjoint) we produce all the individuals of the composite
sets, each unimg' operation being exhaustively forced makes a

208

number of memory references proportional to the cardinality
of the RIM set of the relation in question (i.e., R or S).
In defining the algorithms related with each case of function
"Force" we have shown that the above expression (composite
set) could be executed in linear time, because of the way we
define the algorithm for the "Set intersection" operation.
So the above algorithm also has the worst case asymptotical
time complexity behaviour of O(n) where "n" is the maximum of
the RIM set cardinalities of the relations R and S.

(non:R)c:x

The algorithm for this case is as follows:

1. Force the composite set:

((rim:R) - (unimg':R:x))

once; get the individual returned and return it.

Now we will do the worst case asymptotical time
complexity analysis of this algorithm.

In the worst case the set (rim:R) becomes exactly equal
to the set (unimg':R:x), in this case we have to produce all
the individuals of the set (rim:R) in addition to the
individuals of the set (unimg':R:x). So we can write the
complexity function of this algorithm as follows:

$f = K*n + L*n + C$

where:

K = The constant number of memory references made for
    obtaining each individual of the set (rim:R).

L = The constant number of memory references made for obtaining each individual of the set, (unimg':R:x).

C = The constant number of overhead memory references.

n = The cardinality of the RIM set of the relation R.

By looking at the above function we conclude that the worst case asymptotical time complexity behaviour of this algorithm is O(n), where "n" is the cardinality of the RIM set of the relation.

The change in the time complexity behaviour of some algorithms in the case the relations are restricted to be injective will be inspected next.

If we restrict the operand relations of some composite relations to be the injective relations, the function application operation makes a constant number of memory references while working on those composite relations.

Function application algorithms on those composite relations are as follows:

(R&S):x

 1. Apply R to the "x", save the individual returned.

 2. Apply S to the "x", save the individual returned.

 3. Compare the individuals saved in steps 1 and 2; if they are the same, return this individual; otherwise call the error routine.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As we know the function application operation on the extensionally represented relations has a constant time behaviour so in steps 1 and 2 we make a constant number of memory references. Because step 3 does only one comparison we conclude that the worst case asymptotical time complexity behaviour of this algorithm is O(c). Note that this is true in the case the relations R and S are extensionally represented relations.

(R-S):x

1. Do steps 1 and 2 of the algorithm defined for the composite relation (R&S) above.

2. Compare the individuals saved; if they are not the same, return the individual obtained by applying the relation R to the argument individual "x"; otherwise call the error routine.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The algorithm has the same asymptotical time complexity behaviour as the algorithm defined for the composite relation (R&S), because the same argument applies.

F. ALTERNATIVE METHOD FOR GENERATING INDIVIDUALS OF COMPOSITE SETS

In defining the algorithms for the cases of the force primitive we used the unimg: operation as a primitive operation. When the unimg: operation works on an

211

extensionally represented relation, we know that the algorithm for the unimg: operation has linear time complexity behaviour. On the other hand, if the relation is a complex compound relation, this primitive operation may cost us more. For example, the unimg: operation on the compound relation RS (where R and S are extensionally represented relations) has a worst case asymptotical time complexity behaviour of $O(n^2)$. We defined the algorithms of the cases of the force primitive for the operator pairs that must be specially treated and we had a total of 27 algorithms.

In defining the alternative algorithms for the cases of the force primitive we will define an algorithm for each operator rather than each operator pair. This may seem to the reader more efficient than our previous method and the reader may naturally think that by using this method we will reduce the number of cases that we have to define algorithms for, but this is not true in our case. In defining the algorithms for our interpreter, we mentioned the concept of generalization, and we defined our algorithms in terms of five primitive operations. It turned out that we could reduce some compound sets to the other kinds of compound sets in the preprocessing phase. Hence we could express some composite sets which are constructed by using a relational operation in terms of the other composite sets that are constructed by using one or more of the five primitive

relational operations. So we did not have to include the algorithms for the operator pairs involving that kind of relational operators that is reducible in the preprocessing phase. This allowed us to reduce the number of cases of the "Force" primitive to 27.

In the second method we produce the tuples of the compound relations and do the primitive operations on these tuples by defining an algorithm for each operator that constructs a compound relation and by defining an algorithm for each primitive operator (i.e., !:, unimg:, lem:, rim:) which yields a compound set when combined with a compound relation. Because any operand relation in a compound relation may be another compound relation, we have to define an algorithm for each of the operators which constructs a compound relation and which was reducible in the preprocessing phase (in the case of the previous method used). In addition some of the cases of the "Force" primitive has to be included in the cases defined for the new method, so we will have totally 36 algorithms for the 36 cases defined for the new method instead of 27 cases defined for the previous method. The first question we have to ask ourselves is: What are the efficiencies associated with the new method that motivate us to investigate it? The important efficiency of the new method is no matter how complex a composite set is, the operation producing the individuals of

213

this set has the worst case asymptotical time complexity behaviour of $O(n^2)$. We will make this fact clear in defining the algorithms for the operators. In our previous method, depending on the cost of unimg: operation on the compound relation in question, the cost of an algorithm defined for a case of the "Force" primitive can increase arbitrarily, but in this case the asymptotical time complexity behaviour of an algorithm is fixed. On the other hand all of the algorithms defined for the new method have the worst case asymptotical time complexity behaviour of $O(n^2)$, while we had some $O(n)$ algorithms with previous methods, note that these algorithms have a linear time complexity behaviour in the case the operand relations of the compound sets are represented extensionally. Again the cost of these algorithms may increase arbitrarily depending on the kind of compound relations and the cost of the unimg: operation on those compound relations.

In our new method we will define a new primitive function, namely "Force 2", which works exactly the same as the "Force" primitive. The distinction is, it produces the tuples of the compound relations instead of the individuals of compound sets. The state saving mechanism works almost in the same way as we defined for the "Force" primitive, but in this case we have two pointers to be saved instead of one. Each case of this primitive corresponds to a relational

214

operation that constructs a compound relation out of the operand relations. The "Force 2" primitive is forced with a compound relation being the argument. Then the appropriate case of the "Force 2" primitive is invoked and the algorithm defined for this case divides this compound relation into simpler compound relations and calls the "Force 2" primitive recursively with each of the compound relations created being the argument (i.e., it calls the "Force 2" primitive with only one compound relation at a time). This process continues until an extensionally represented relation is forced, in which case the algorithm below applies:

1. Hash into the relation table with the relation identifier; find the record of the relation.

2. Follow the pointers found in the PFLM and PFRM fields of this record and find the records of the first left member and the first right member.

3. Put the pointer to the first left member's record into the left field of the record structure to be returned and put the pointer to the first right member's record into the right field of the record structure to be returned. (* The tuples are returned to the higher levels by using the record structure shown in Figure 16. *)

left	right
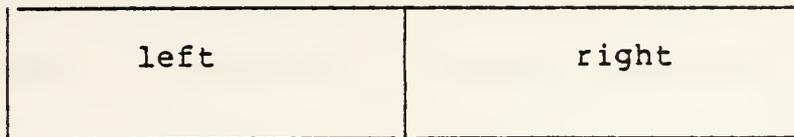
Figure 16.   The conveying record structure.

4.   Take the next integer from the global count associated with the high level compound relation, advance the pointer which is currently pointing at the first right member's record to the next record of the RIM set, and put this pointer into the right field of the record (which has the above structure) which will be saved in the hash table associated with the "Force 2" primitive.   Put the pointer pointing at the current LEM set record into the left field of this record.   Hash to the hash table which we will call MHASH 2, with the integer obtained above being the identifier, and save the above record in this hash table under this identifier.   (* The high level compound relation mentioned above is analogous to the high level compound set that we defined in the "Force" primitive case. *)

5.   If a subsequent force is addressed to this relation associated with the same high level compound relation, take the next integer from the global count associated with this high level compound relation (which should be the same as the integer found in step 4), hash to the MHASH 2 table with this

integer as the identifier and return the record found under this identifier. Decrement the global count and repeat step 4.

6. If in any force operation the pointer proceeding in the RIM set of the relation reaches the end of the RIM set and can not proceed further, reset this pointer to the beginning of the RIM set and advance the pointer pointing at the current record in the LEM set to the next record of the LEM set. If the LEM set is also exhausted, return "nil" to the caller (which is the function Force 2 itself) and save 0's in the left and right fields of the record associated with this level instead of pointers.

Hence the case associated with the extensionally represented relations returns the tuples of the relation in question, one at a time as "Force 2" is forced repeatedly with the same relation identifier as the argument. If a tuple does meet the conditions imposed by the cases that are involved in the path of recursion, it is returned by the "Force 2" primitive as a tuple of the high level compound relation.

Now we will define the algorithms for the cases of the "Force 2" primitive. In these algorithms we will refer to the membership test algorithms that have not been defined yet. We will define those algorithms later. Those algorithms will be associated with the operators that

217

construct compound relations. The membership test operations on relations are done in exactly the same way that we explained in the compound sets case; i.e., given a relation and a tuple to be tested, the algorithms defined for the operators divide the membership test task into simpler membership test tasks by calling each other until the membership test/tests can be done on an extensionally represented relation, as we described in chapter 1. The membership test can be done in constant time on an extensionally represented relation. Because many of the membership test algorithms divide the membership test task into simpler membership test tasks in constant time, no matter how complex the initial compound relation is, the membership test operation can be done in constant time on most of the compound relations. We will explain some algorithms as if "Force 2" is being forced repeatedly but for others we will specify the action for only one force operation, in order to make the algorithms clear to the reader.

R&S:

The algorithm for this case can be defined as follows:

1. Force the relation R, get the tuple returned.

2. Test if this tuple in relation S; if so return this tuple, else go to step 1.

218

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We know that in the worst case a relation may have at most (n^2) tuples when "n" is the common cardinality of the domain and codomain of the relation. So producing all the tuples of a relation requires at least a number of memory references proportional to the square of n, independent of the underlying representation technique. So, because the above algorithm produces all the tuples of one of the operand relations (possibly a compound relation) it requires (at least) a number of memory references proportional to the square of "n".

Now suppose R is a compound relation. If we can produce the tuples of this compound relation in $O(n^2)$ time, we would not care about whether R is a compound relation or an extensionally represented primitive relation in deciding the asymptotical time complexity of R&S, because in either case we are producing the tuples in $O(n^2)$ time. Let's suppose that R is defined as T&D, where T and D are compound relations. If we can produce the tuples of T in $O(n^2)$ time, we would not care about whether T is an extensionally represented relation or a compound relation. So if we can produce the tuples of each kind of compound relations (i.e., R-S, R&S, R#S, and so on) in $O(n^2)$ time, no matter how complex the initial compound relation is, we can produce its

tuples by making a number of memory references proportional to the square of "n". Of course after a certain number of nesting levels and in some instances, the value corresponding to the constant multiple of the square of "n" may be much larger than the value of the cube of "n", but the asymptotical time complexity behaviour of the operation on the initial compound set, is still $O(n^2)$.

In deciding about the asymptotical time complexity of the above algorithm we assumed that the membership test operation (in step 2) can be done in constant time, but this is not always true. In these algorithms we will continue to assume the membership test operation as a constant time operation. There are some compound relations for which we can not define $O(n^2)$ algorithms for producing their individuals, such as fan:R and san:R. We will discuss the effects of these drawbacks later.

R|S

The algorithm for this case can be defined as follows:

1. Force the relation R, get the tuple returned.

2. Test if this tuple is in relation S. If so go to step 1 else return this tuple.

3. After no more tuple remains to be returned from the relation R as a result of the repeated force operations, force the relation S, and return the tuples of S one at a time as the compound relation R|S is forced.

220

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The same time complexity analysis that we have done in the compound relation R&S case applies to this compound relation also.

R-S

The algorithm for this case can be defined as follows:

1. Force the relation R, get the tuple returned.

2. Test if this tuple in relation S; if so go to step 1 else return this tuple.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The same time complexity analysis that we have done in the compound relation R&S case applies to this compound relation also.

R||S

The algorithm for this case is a very expensive algorithm. We will only explain how expensive it is and why it is expensive.

Producing all tuples of this compound relation implies producing all individuals of the Right Members set. As we know, the Right Members Set of the compound relation R||S is equal to the cartesian product of the RIM set of R and The RIM set of S. As we explained before, obtaining the individuals of a composite set (in the new system) requires a

221

number of memory references proportional to the square of n, where n is assumed to be the common cardinality of the domain and codomain of the relation in question. So we obtain the individuals of rim:R in $O(n^2)$ time and for each individual obtained we produce all the individuals of rim:S in $O(n^2)$ time. So assuming the cardinalities of the RIM sets and LEM sets of the relations R and S are equal to n we conclude that we produce the individuals of rim:$(R||S)$ by making the number of memory references proportional to (n^4). In this algorithm we create a pair relation out of each pair of individuals obtained in the manner explained above (by producing the individuals of rim:R and rim:S). We apply the relation R to the right component of this individual (which is a pair) and we apply relation S to the right component of this individual, then we pair the resulting individuals up. Note that this function application operation may cost us $O(n)$ time which may make the algorithm's time complexity behaviour $O(n^5)$.

(R#S)

The algorithm for this case can be defined as follows:

1. Force the rim:R, get the individual returned.

2. Apply the relation R and relation S to this individual, pair the resulting individuals up (i.e., establish a record for pairs in the relation table), and put the pointer to the record of this pair in the left field of

222

the record to be returned. In the same manner, establish the
pointer (in the right field of the record to be returned) to
the memory location where the individual obtained in step 1
is saved.

3. Return the record (tuple) obtained in step 2 to the
caller.

Now we will do the worst case asymptotical time
complexity analysis of this algorithm.

As we stated before, in most of the cases, we can obtain
the individuals of the RIM set of a relation in $O(n^2)$ time.
In the above algorithm, for each individual we obtained, we
call the function application algorithm two times. As we
know, most of the function application algorithms have linear
time complexity behaviour. Assuming the function application
algorithms (in this case) have a linear time complexity
behaviour and the cardinality of the rim:R, lem:R and lem:S
are each equal to n, we conclude that the algorithm has the
asymptotical time complexity behaviour of $O(n^3)$. Note that
some of our function application algorithms have used the
unimg: operation on compound relations. That means this
algorithm will have quadratic behaviour in the new system, so
in the worst case we call a quadratic algorithm for each
individual of the rim:R obtained. So the worst case
asymptotical time complexity behaviour of the algorithm is
in fact $O(n^4)$, which is again an expensive algorithm.

Rc

The algorithm for this case can be defined as follows:

1. Force the relation R, get the tuple returned.

2. Switch the components of this tuple and return the resulting tuple. (* That means put the pointer to the right component of the tuple (which belongs to R) into the left field and left component of the tuple into the right field of the record to be returned. *)

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As we can see, the algorithm does not do more than two assignments, so it has the worst case asymptotical time complexity behaviour of $O(c)$.

RS

The algorithm for this case can be defined as follows:

1. Force the relation S, get the tuple returned.

2. Apply the relation R to the left component of this tuple, take the resulting individual.

3. Put the pointer to this individual (actually the memory location where this individual is saved) in the left field and the pointer to the right component (individual) of the tuple obtained in step 1 into the right field of the record to be returned.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

224

As we know, if the above algorithm is forced repeatedly, step 1 costs us a number of memory references proportional to the square of n, where n is assumed to be the common cardinality of the rim:S and lem:S. For each tuple obtained in step 1, we call the functional application algorithm. Assuming the function application algorithm has the linear time complexity behaviour and the lem:R has the cardinality n, we conclude that the algorithm has the time complexity behaviour of $O(n^3)$. If the function application algorithm in question has the constant time complexity behaviour, the algorithm would have the time complexity behaviour of $O(n^2)$. Note that in our new system, the function application algorithms using the unimg: operation on compound relations, will automatically have quadratic behaviour. This in fact causes the worst case asymptotical time complexity behaviour of this algorithm to be $O(n^4)$.

Now we will define the algorithms that will continue to be maintained in our Force primitive and are associated with our five basic operations.

unimg:R:x

The algorithm for this case can be defined as follows:

1. Force the Force 2 primitive with relation R being the argument; get the tuple returned.

2.   If the right component of this tuple is equal to the individual x, take the left component of this tuple and return it.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As can be seen the algorithm makes a comparison (taking a constant number of memory references) for each tuple obtained and we know that the tuples of a relation are produced by making a number of memory references proportional to at least the square of n, where n is assumed to be the common cardinality of the rim:R and lem:R. So the time complexity behaviour of the algorithm is at least $O(n^2)$.

R!:C

The algorithm for this case can be defined as follows:

1.   Take the next integer from the global count.

2.   Force the set C repeatedly. For each individual obtained in this manner, hash to the LHT using this individual with the integer obtained in step 1 as the relation identifier. Establish the record of this individual in the LHT. Link the records created in this manner to each other as they are created.

3.   Force the relation R repeatedly. For each tuple obtained, extract the right component individual and hash with this individual into the LHT using the integer obtained in step 1 as the relation identifier. If there exists a

226

record for this individual in the LHT, do step 4; else force the relation R in order to obtain the next tuple.

4.   Extract the left component individual of the current tuple, hash with this individual into the RHT with the integer obtained in step 1 as the relation identifier, and establish the record of this individual in the RHT under this relation identifier if there is no record for this individual in the RHT.   Link the records created in this manner to each other as they are created.

5.   After no more tuples remain to be produced in the relation R,   return the first individual of the set constructed in the RHT (in step 4).   In the repeated force operations, return the individuals of this set one at a time.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We can write the time complexity function of this algorithm as follows:

$$f = K*(n^2) + L*(n^2) + M*n + C$$

where:

K = The constant number of memory references made in each
     iteration of step 2.

L = The constant number of memory references made in
     order to obtain each tuple of the relation R in step
     3.

M = The constant number of memory references made in step
    5 in order to return each individual of the set
    obtained (in step 4).

C = The constant number of memory references made by the
    remaining steps.

n = The common cardinality (assumed) of the LEM and RIM
    sets of all relations involved. (If the compound set
    C is defined in terms of relations and relational
    operations, the LEM and RIM set cardinalities of
    these relations are also equal to n).

In the above algorithm we assumed that the individuals of the
argument set C are producible by making a number of memory
references proportional to the square of n. Because the
argument set C may be a compound set, we choose the typical
complexity behaviour of the operation producing the
individuals of C as $O(n^2)$. In the above complexity function,
the first term corresponds to step 2, the second term
corresponds to step 3, the third term corresponds to step 5
and the last term corresponds to the remaining steps. As we
can see, the algorithm produces the resulting set in the RHT
when it is forced for the first time, and in step 5 we are
returning the individuals of an extensionally represented set
one at a time, so the term corresponding to step 5 has linear
behaviour. We determine the asymptotical time complexity
behaviour of this algorithm by looking at the term of the

228

complexity function which has the highest exponent; hence we get $O(n^2)$. If it turned out that we produce the individuals of the set C or the tuples of the relation R in time proportional to (n^3), the algorithm automatically becomes an $O(n^3)$ algorithm.

lem:R

The algorithm for this case can be defined as follows:

1. Force the relation R, get the tuple returned.

2. Take the left component individual of the tuple and return it.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The asymptotical time complexity behaviour of this algorithm is the same as the asymptotical time complexity behaviour of the operation which produces the tuples of the relation R; i.e., if we are producing the tuples of the compound relation R in time proportional to the square of n, the time complexity behaviour of this algorithm becomes $O(n^2)$.

rim:R

The algorithm for this case is very similar to the algorithm for the lem:R; the only difference is we take the right component individual instead of the left component individual in step 2.

229

G. ADDITIONAL MEMBERSHIP TEST ALGORITHMS FOR THE ALTERNATIVE METHOD

In this section we will define the algorithms for the membership test corresponding to the relational operators that construct compound relations. Because a relation can be viewed as a set of tuples and the tuples can be viewed as the members of this set, we will continue to use the term "membership test".

R&S

The algorithm for this case can be defined as follows:

1. Test if the given tuple is in relation R; if so do step 2 else return false.

2. Test if the given tuple is in relation S; if so return true else return false.

Now we will do the worst case asymptotical time complexity analysis of this alogrithm.

If we assume the relations R and S are extensionally represented relations, then step 1 costs us a constant number of memory references, and similarly for the second step. So in this case the algorithm has the worst case asymptotical time complexity behaviour of $O(c)$. Now suppose the relation R is a compound relation defined as T&H, where T and H are extensionally represented relations. Since the membership test operation on this compound set also requires the constant number of memory references to be made, we can view

230

T&H as an extensionally represented relation for just this purpose. So if we can define a constant time algorithm for each kind of compound relation (i.e., R&S, R-S, Rc, R|S, etc.) no matter how complex the initial compound relation is we can do the membership test in constant time. But not all kinds of compound relations can be associated with constant time membership test algorithms; we will discuss the effects of this inefficiency later.

R|S

The algorithm for this case can be defined as follows:

1. Test if the given tuple is in relation R; if so return true, else do step 2.

2. Test if the given tuple is in relation S; if so return true.

Now we will do the worst case asymptotical time complexity analysis of this alogrithm.

The argument that we have done in the compound relation R&S case applies to this compound relation also; i.e., the algorithm has the asymptotical time complexity behaviour of O(c).

R - S

The algorithm for this case can be defined as follows:

1. Test if the given tuple is in relation R; if so do step 2, else return false.

231

2.   Test if the given tuple is in relation S; if so return false, else return true.

Now we will do the worst case asymptotical time complexity analysis of this alogrithm.

The argument that we have done in the compound relation R&S case applies to this compound relation also; i.e., the algorithm has the asymptotical time complexity behaviour of $O(c)$.

Rc

We can define this algorithm as follows:

1.   Switch the component individuals of the given tuple, test if the resulting tuple is in relation R; if so return true else return false.

Now we will do the worst case asymptotical time complexity analysis of this alogrithm.

The argument that we have done in the compound relation R&S case applies to this compound relation also; i.e., the algorithm has the asymptotical time complexity behaviour of $O(c)$.

non:R

We can define this algorithm as follows:

1.   Test if the given tuple is in relation R; if so return false else return true.

Now we will do the worst case asymptotical time complexity analysis of this alogrithm.

232

The time complexity behaviour of the algorithm is the same as the time complexity behaviour of the algorithms given above and the same argument applies.

R||S

The algorithm for this case can be defined as follows:

Let the given tuple be <(a,c), (b,g)> where (a,c) and (b,g) are pairs (individuals).

1. Test if the tuple <a,b> is in relation R; if so do step 2 else return false.

2. Test if the tuple <c,g> is in relation S; if so return true else return false.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As can be seen the algorithm makes two membership tests. If both membership test operations have the constant time asymptotical time complexity behaviour then the algorithm has the asymptotical time complexity behavior of O(c). If any one of the membership test operations has a time complexity function which dominates the constant function, then the algorithm has the same asymptotical time complexity behaviour as the time complexity behaviour of this operation.

R‡S

We can define this algorithm as follows:

Let the given tuple be <(a,c),b> where (a,c) is a pair (individual).

1. Test if the tuple <a,b> is in the relation R; if so do step 2 else return false.

2. Test if the tuple <c,b> is in the relation S; if so return true.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The algorithm has the same asymptotical time complexity behaviour as the asymptotical time complexity behaviour of the algorithm defined for R||S.

RS

In this algorithm we have to use the unimg': operation in order to determine if the given tuple is in the relation RS, because the left component of the given tuple should be a member of the LEM set of the relation R and the right component of the tuple should be the member of the RIM set of the relation S. Hence we can not easily determine if the given tuple is in the relation RS, especially in case the relation R and S are themselves compound relations. The algorithm is as follows:

1. Test if the right component individual of the given tuple is in the rim:S; if so do step 2 else return false.

2. Force the composite set unimg:R:x repeatedly, store the individuals of this set in the RHT as it has been done before in previous algorithms.

3. Force the unimg':R:y repeatedly where y is the left component individual of the given tuple. For each individual produced, test if this individual in the set established in the RHT above; if so return true, else continue testing the next individual of the unimg':R:y.

4. If neither of the individuals of the unimg':R:y is in the set established in the RHT, return false.

Now we will do the worst case asymptotical time complexity analysis of this alogrithm.

As can be seen the algorithm is an expensive algorithm relative to the other membership test algorithms. We know that producing the individuals of the unimg:R:x and the unimg':R:y both requires a number of memory references proportional to the square of n, where n is assumed to be the common cardinality of the LEM and the RIM sets of the relation R. So the algorithm has to make at least (n^2) memory references. We conclude that the algorithm has the asymptotical time complexity behaviour of $O(n^2)$. Of course, if the complexity of the operations unimg:R:x and unimg':R:y were higher, the complexity of this algorithm would increase.

The membership test algorithms on the compound relations fan:R and san:R are much more costly than the above algorithm; hence we will construct the extensional representation structure (in less time) for these compound relations in order to be able to do the membership test. If

235

we also construct the extensional representation structure
for the compound relation RS whenever a reference to this
compound relation occurs for the first time, we can assert
that: Given that the compound relations fan:R, san:R and RS
are represented extensionally, no matter how complex the high
level compound relation is, the membership test on this
compound relation can be done in constant time.

H.   COMPARISON OF THE TWO METHODS

We stated that, in the first method, the algorithms may
get arbitrarily expensive depending on the complexity of the
compound relation or set in question.   On the other hand, in
the second method, if we construct the extensional
representation for the kind of compound relations which are
associated with the expensive production (Force 2 cases) or
membership algorithms, we can do the operations in $O(n^2)$ time
independent of the complexity of the compound relation or
compound set in question.   Let's now assume that we did not
construct the extensional representations for compound
relations, R#S, R||S, fan:R and san:R, and let's assume the
relations R and S are represented extensionally.   In this
case the operation (R#S)!:C has the asymptotical time
complexity behaviour of $O(n)$ by using the first method and if
we use the second method it has the asymptotical time
complexity behaviour of $O(n^4)$.   The Table 1 shows the
differences in asymptotical time complexity between the two

236

Table 1. The asymptotical time complexity behaviour of various algorithms under method 1 and method 2.

	METHOD 1	METHOD 2
R!:C	$O(n^2)$	$O(n^2)$
(R-S)!:C	$O(n^3)$	$O(n^2)$
(R#S)!:C	$O(n)$	$O(n^4)$
(R||S)!:C	$O(n)$	$O(n^4)$
(RS)!:C	$O(n^2)$	$O(n^4)$
unimg:R:x	$O(n)$	$O(n^2)$
unimg:RS:x	$O(n)$	$O(n^4)$
lem:R	$O(n)$	$O(n^2)$
lem:(R-S)	$O(n^3)$	$O(n^2)$
lem:(R#S)	$O(n)$	$O(n^4)$
lem:(R||S)	$O(n^2)$	$O(n^4)$
lem:RS	$O(n^2)$	$O(n^4)$
Rc!:C	$O(n^2)$	$O(n^2)$
(R#S)c!:C	$O(n^2)$	$O(n^2)$
unimg:(R#S)c	$O(n^2)$	$O(n^4)$
unimg:(R||S)c	$O(n^2)$	$O(n^4)$

methods. In this table we also show the differences in asymptotical time complexity in the case when we represent some of the compound relations extensionally (in the second method).

We should not consider the second method more efficient than the first method because when we produce the individuals of a compound set by making no less than (n^2) memory references, our function application algorithms and set membership test algorithms become automatically $O(n^2)$ algorithms, while they were $O(n)$ algorithms in the case method 1 was used. Again we can not forget that those linear function application and set membership test algorithms exist in the case the operand relations of the compound relations are represented extensionally. The cost of those algorithms may go up depending on the complexity of compound relation or set in question. Note that we can reduce the overall complexity of the operations in the first method by representing some kind of compound relations, like R-S, RS, fan:R, etc. extensionally like we have done in the second method. As a criterion we can say that, if the nesting levels in our compound relations do not exceed 1 or 2, the first method should be used; otherwise the second method is more appropriate.

I. THE WAY THE SYSTEM HANDLES THE RESTRICTION OPERATIONS

The restriction operations, "Right restriction", "Left restriction", and the "Restriction", are handled by the system in a special manner. The operations that we defined before are done in exactly the same manner as defined on the restricted relations also, but any individual obtained from the LEM set of a left restricted or restricted relation is tested for membership in the set to which the LEM set of the relation is restricted. In the same manner any individual obtained from the RIM set of a right restricted or restricted relation is tested for membership in the set to which the RIM set of this relation is restricted. Hence the system treats the restriction operations as general operations.

This feature of the system can be implemented by defining a seperate routine which is given a set (possibly a composite set represented in character string form) and an individual drives the appropriate routines defined for the membership test in order to test the membership of this individual in the given set. This routine can be called by any operation that has just obtained an individual from the LEM or RIM set of a left restricted and/or right restricted relation, so this operation waits for positive response from this routine and upon getting the positive response (true), the operation may do whatever it intended to do with the individual in question.

239

So as a result, if the system is implemented, this feature should be integrated with each algorithm which we defined earlier, whenever it is applicable.

IV. THE PURE INTENSIONAL REPRESENTATION SYSTEM

We defined the primitive relations as the relations defined by the user, which may be represented either extensionally or intensionally. In the same manner we defined the primitive sets as the sets defined by the user, which may be represented either extensionally or intensionally. The system we defined assumed that the primitive relations and sets are represented extensionally. Now we will think about how the system can be adapted to the case in which the primitive relations and sets are represented intensionally; in other words when we have an expression representing a primitive relation or a set rather than a data structure.

In defining the algorithms for our system, we focused on three main groups of algorithms, namely:

1.   The algorithms for the production of the individuals of the intensionally represented composite sets.

2.   The algorithms that do the membership test on the intensionally represented composite sets.

3.   The algorithms for the function application operation on each kind of intensionally represented composite relation.

We will focus on these three groups of algorithms in the case the primitive relations and sets are represented intensionally.

In defining the mechanism for the production of the individuals of the intensionally represented composite sets, we designate five basic operations and we reduce the operations on those composite sets to these five basic operations on the extensionally represented relations. So if we can define these operations on the intensionally represented primitive relations we can adapt our individual production mechanism to the case in which the primitive relations are represented intensionally. Because the system does not pay attention to the way the primitive relations are represented until one of the five basic operations is done on the primitive relation, we can adapt the system to this case by only defining the algorithms for the five basic operations on the intensionally represented primitive relations. But this is not as easy as it seems at first glance; first of all if a relation is not a function, we can not easily define the code which represents this relation. On the other hand if we restrict our relations to the functions, three of the five basic operations, Unit image, Unit coimage, Image, become undefined and the remaining operations, Left members and Right members, are hard to define on the intensionally represented relation because in some cases the domain of a

242

relation may be an infinite set. Producing the individuals of this set brings the question, "How many of the individuals will be produced?" and having the user declare the interval of input values (domain individuals) for his function is not logical and has not been done in any language. So the mechanism for the production of the individuals of the composite sets when the primitive relations are represented intensionally, is hard to define and even if it is defined it brings many undesirable restrictions to the user.

In defining the above system we could be able to define less costly algorithms for membership tests on some kind of composite sets, but for some of them we had to do the membership test in a produce and test fashion. Hence there is no point in defining the whole mechanism for our new case by knowing the fact that we will not be able to define algorithms for some composite sets. So the mechanism for membership test is not a well defined mechanism when the primitive relations are intensionally represented. The prime reason which causes that is the "Image" operation which requires the argument set individuals explicitly in order to accomplish its job. We know that the mechanism for the production of the individuals of the intensionally represented composite sets is not a well defined mechanism in the case the primitive relations and sets are represented intensionally.

The algorithms that fall into the last category are well defined in the case the primitive relations are represented intensionally, but these relations should be restricted to be the functions. So in that mechanism, the user can define his functions in advance by using a high level language and by compiling them, then he may introduce those routines into our system by linking them to the system. During this linking operation, the records of those relations (functions) are established in the relation table with the user defined identifiers being the relation identifiers and pointers to the related codes are established in the PCOLS fields of those records. So whenever the function application operation is to be performed on an intensionally represented primitive relation (user defined function), the system finds the record of this relation; extracts the pointer to the code from the PCOLS field of this relation's record and calls the function with the argument individual.

So we have seen that the structure of the system allows the user to define functions and embed them in the system, but it does not allow the user to use his/her functions in defining the composite sets. As an example, suppose the user defined a function called, "+", which is given an integer returns the successor of this integer. He may use this function in any expression as long as the operation to be done on this relation is function application.

V. <u>CONCLUSIONS</u>

In this thesis we tried to find out efficient ways to represent binary relations that make the algorithms of the relational operations efficient in time. We did this by inspecting the worst case asymptotical time complexity behaviour of the algorithms.

The first representation techniques that we inspected were the extensional representation techniques. Among them we selected the Incidence Matrix representation and used it in combination with hash tables; we called the resulting representation technique the Hash-Incidence-Vector representation. This representation technique enabled us to create efficient algorithms relative to the algorithms defined for the Table representation. We have observed that among the 25 relational operation algorithms defined for the Hash-Incidence-Vector representation, two are constant time algorithms, twelve are $O(n)$ time algorithms, seven are $O(n^2)$ time algorithms, three are $O(n^3)$ algorithms and one is $O(n^4)$ algorithm. Among the $O(n^2)$ time algorithms, only three have been observed to be very expensive because their time complexity functions had large constants in front of the second degree terms. The other $O(n^2)$ time algorithms have

been observed to be cheaper, because the constants in front of the second degree terms of their complexity functions are less than 1.

As a result, the operations, Relation Intersection, Relation Difference, Relation Union, Relation Composition (Relative Product), Parallel Application, First Ancestral and Second Ancestral, have been found to be expensive operations. Among them, Relation Intersection, Relation Difference, Relation Union and Parallel application operations can be associated with constant time and $O(n)$ time algorithms in the case the intensional representation techniques used. The algorithms for the Relative Product, First Ancestral, and Second Ancestral operations, have $O(n^3)$ worst case asymptotical time complexity behaviour. However, in their time complexity functions the constants in front of the third degree terms are less than 1, which makes these algorithms executable for small n's (100-200).

As can be seen, most of the algorithms have a worst case asymptotical time complexity behaviour of $O(n)$. Why couldn't we define more efficient algorithms? The first reason is most of the operations must examine all of the individuals of the sets involved. The second reason is we want to save the original relations and sets while constructing new relations and sets out of the original relations and sets. This requires extensive copying operations and causes most of the

246

algorithms to have a worst case asymptotical time complexity behaviour of O(n).

In the intensional representations case, we observed that the algorithms become expensive in time but that we save a lot of space by not constructing the extensional representation structures for the intermediate sets and relations in the memory. On the other hand, we have been able to define cheaper intensional algorithms for some of the operations that are associated with expensive extensional algorithms (such as parallel application).

In Chapter IV we have seen that the pure intensional representation mechanism is not a well defined mechanism but we are able to include user defined functions if we restrict the use of those functions to the function application operation.

So, the extensional representation techniques enable us to define time efficient algorithms and the intensional representation techniques enable us to define space efficient algorithms. For us, using both representation techniques in combination with each other (rather than firmly selecting one of them) is necessary. If we are to define a criterion for establishing this combination, we would use intensional representation techniques for Relation Intersection, Relation Union, Relation Difference and Parallel Application operations. We would use extensional representation

247

techniques for Relative Product, First Ancestral and Second Ancestral operations when they are involved in a composite set construct; otherwise, we would use intensional representation techniques for these operations. The above criterion can be refined by taking into account the available hardware features. For example, if we have a limited memory we would use the intensional representation techniques for most of the operations.

As can be easily seen, it is feasible to implement the language on conventional architectures. But it would be nice to have an architecture which supports this language. This architecture has to have at least these properties:

1. It has to support hash coding.

2. It has to have pipelining, or an equivalent mechanism, which has at least an ORing stage.

3. It has to have a mechanism to speed up copying operations.

4. It has to support bit string and character string data types.

As we indicated before, the efficiency of most of our algorithms increases as the memory word length increases. So as long as we can fetch more bits for each memory cycle, the speed of our algorithms increases proportionally. Thus we must be careful about the word length if we use the Hash-Incidence-Vector representation.

We conclude that it is feasible to implement this language on conventional architectures and that we can make full use of this powerful language by having more suitable architectures.

THE EXTENSIONAL ALGORITHMS CONTINUED

Image Operation (R!:x) :

This operation, given a set C, produces the set of individuals in which each individual is in relation with at least one individual in the given set C under the relation that is being applied to the set C. This means the operation is effectively performing the "Unit Image" operation on each individual of the given set C, then performing the set union operation on the resulting sets to obtain the set in which each individual is in relation with at least one individual of the set C. We can state this more carefully: Let R be a relation and C be a set, then R!:C is the set of all y such that yRx for some x in C.

The algorithm for Hash-Incidence-Vector representation is as follows:

1. Get the relation identifier and the set identifier.

2. Hash with the relation identifier to the relation table and find the record of the relation, follow the pointers found in the PFLM and in the PFRM fields of that record and find the records of the first left member and the first right member respectively.

3. Hash with the set identifier to the set table, find the record of the set, follow the pointer found in the PSS field of that record and find the first record of the set

structure. Start from the beginning of the linked list
structure of the set and proceed down in that structure
record by record. For each record found in this manner hash
with the individual being represented by that record to the
RHT under the given relation identifier. If the RHT record
of that individual is present in the RIM set of the relation,
extract the index of the RHT record that represents that
integer and hash into the RHT with that individual under the
relation identifier:

   "$$$"

and establish the RHT record of that integer. Link the
records created in this manner in the RHT by their TASE links
as they are created.

   4.   Start from the beginning of the LEM set of the
relation, proceed down in the LEM set record by record. For
each record found, extract the index of the record. Put it
in an index register and increment it up to the number:

   INDEX + cardinality of the RIM set of the relation

by beginning with 1. For each increment hash into the RHT
with the integer:

   CURRENT VALUE - INDEX

under the relation identifier "$$$". If a record for the
resulting integer is found to be in the RHT, call this

integer J, reference the incidence vector of the relation with the index

    K = J-1

by calling the "Reference" algorithm. If there is a 1 in the corresponding incidence vector location, hash into the SHT with the left individual' being represented by the current LEM set record and establish the record of that individual. Link the SHT records created in this manner to each other as they are created. Keep a count and increment that count for each record created in the SHT by beginning with 0. As soon as a 1 is found for a left individual in the above manner, quit with that left individual and continue to perform the above process for the next individual of the LEM set, by following the TASE link of the current left individual's record and finding the next individual's LHT record in the linked list structure of the LEM set of the relation.

    5.   Start from the beginning of the linked list of temporary records created in the step 3, (in the RHT). Proceed down in that linked list structure and for each record found, hash to the RHT with the integer (index) being represented by that record under the relation identifier "$$$", and disconnect it from the RHT entry if it is directly connected to that RHT entry.

6. Hash to the set table with the set identifier:

(relation identifier)'!:'(argument set identifier)

This means, if the relation being applied has the identifier R and the argument set identifier is C, hash into the set table under the set identifier:

R!:C

Establish the record of that set, put the pointer to the linked list structure established in SHT into the PSS field of that record. Put the last value of the count into the CARD field of that record.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

In the worst case every left individual may be in relation with only one right individual and the record of that right individual may be the last record of the relation's RIM set. In that case, in step 4 we make:

ceiling(n/C)

memory references, where:

C = memory word length.

n = The cardinality of the RIM set of the relation.

In addition to that, even though it is unlikely, the argument set C may be a super set of the RIM set of the relation, so in step 3 we effectively make a separate copy of the RIM set of the relation.

253

So under these considerations and by assuming the cardinalities of the RIM set and the LEM set of the relation are the same ("n"), we write the worst case time complexity function of that algorithm as:

$$f = K*n*ceiling(n/C) + L*n + P*n + D$$

where:

   n = The cardinality of the RIM/LEM set of the relation.

   K = The number of memory references made for each left individual's record found in the step 4.

   m = The cardinality of the argument set.

   L = The number of memory references made for each set record found in step 3.

   P = The number of memory references made for each set record found in step 5.

and:

First term corresponds to the step 4, second term corresponds to the step 3, third term corresponds to the step 5, fourth term corresponds to the other steps of the algorithm. Let n/C be an integer and K/C = V, then the complexity function becomes:

$$f = V*(n^2) + P*n + L*m + D$$

In this algorithm we significantly reduce the average case complexity in step 4. That is, as soon as a 1 for a left individual is found that corresponds to a tuple which has the right individual from the given set C, we quit with

that left individual and continue with the next one in the LEM set. Especially in large relations this may decrease the complexity of the term:

ceiling(n/C)

by some constant. But in order to do that we had to stand for some linear terms in the complexity function, and in some particular cases the strength of one of those linear terms may dominate the complexity of the first term.

But we are concerned with the worst case asymptotical time complexity behaviour of that algorithm. By looking at the exponent of the term with the larger exponent and by assuming:

$L*m < V*(n^2)$

we conclude that the worst case asymptotical time complexity behaviour of that algorithm is $O(n^2)$.

Now we have to define the algorithm for the table representation. The algorithm is as follows:

1. Start from the beginning of the linked list structure of the argument set and proceed down in that linked list record by record. For each record found in this manner, search the individual being represented by that record in the right column of the relation's table by starting from the beginning of the table and by looking up the right individual of each record found while proceeding in the table record by record. Because there is a possibility of a duplication of

the individuals in the right column, do it exhaustively. For each record found to have the individual in question as right individual, hash into the SHT with the left individual of that record under the new set identifier (described in step 6 of the previous algorithm), and establish the record of that individual in the SHT if the record of that individual has not been established in the SHT previously. Link the records created in this manner to each other by their TASE links as they are created. Keep a count beginning with 0 and increment it for each record created in the above manner.

(* Step 2 is called the disconnection operation. *)

2. Do step 6 of the previous algorithm.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As can be seen the algorithm is a costly algorithm; in fact there are some other ways to do it more efficiently. One way is to convert the table representation into an adjacency-list-like representation and establish the table in the SCHT. As we mentioned in the storage requirements analysis for large relations, we may use up a large part of our memory source and, even though it is done temporarily, that may cause the heap to get too large, etc. In fact the resulting analysis would be attributed to the adjacency list representation rather than the table representation if we would have done that.

We write the worst case time complexity function of that
algorithm as follows:

$f = K*m*p + C$

where:

p = Relation size/Table size.

In the worst case the relation may be the universal relation,
so by assuming the cardinalities of the LEM set and the RIM
set of the relation are equal to, say, "n", we may replace
the variable "p" with:

$n*n$

So we rewrite the complexity function as:

$F = K*m*(n^2) + D$

where:

m = The cardinality of the argument set (C).

n = The cardinality of the LEM/RIM set of the relation.

K = The constant number of memory references made for
    each argument set record found in step 1.

D = The constant number of memory references made by the
    housekeeping operations.

Clearly, the first term corresponds to step 1 and the second
term corresponds to the constant number of memory references
made by the housekeeping operations and the number of memory
references made in the step 2. Note that in the worst case
"m" may be greater than or equal to "n". Let's assume the
"m" is a constant multiple of "n" and multiply that constant

with the constant K for making the behaviour of the algorithm clear, then the complexity function becomes:

$f = K*(n^3) + C$

So we conclude that the worst case asymptotical time complexity behaviour of that algorithm is $O(n^3)$ or worse. For example if m is equal to the square of n in some instance, the algorithm behaves like an $O(n^4)$ algorithm.

Complement of a relation (non:R):

The complement of a relation can be defined as the set of tuples that belong to the universal relation on the MEM set of the original relation other than the tuples that are in the original relation. So the incidence vector of the complement of a relation is the incidence vector of the original relation in which all the entries are complemented. In the hash incidence vector representation case, all we have to do is complement the incidence vector as a whole to obtain the complement of the relation in question. But we must not forget that we want to preserve the original relation for possible subsequent references, so we need to make a separate copy of the original relation. The algorithm for Hash-Incidence-Vector representation is as follows:

1. Make separate copies of the LEM set and the RIM set of the original relation under the new relation identifier "non:R".

2. Allocate a block of memory as large as the original incidence vector.

3. Establish the record of the new relation in the relation table, furnish its fields as was done in the previous algorithms.

4. Pipeline the original incidence vector, obtain the complements of the sequence of bits as they fit into the accumulator and copy them to the corresponding location in the new incidence vector.

In this algorithm we assumed the existence of some hardware help (pipelining) but that does not change the asymptotical complexity behaviour of that algorithm. Since it only speeds up the execution by some constant factor, in the absence of pipelining the resulting asymptotical time complexity would be the same. The worst case time complexity function of that algorithm can be written as:

$$f = K*n + L*m + T*((n*m)/Cl*C2)) + C$$

where:

m = The cardinality of the LEM set of the original
   relation.

n = The cardinality of the RIM set of the original
   relation.

The constant K is the number of memory references made while copying each RIM set record; the constant L is the number of memory references made while copying each LEM set record; the

constant T is the number of memory references made while complementing and copying each bit sequence of the original incidence vector; the constant C1 is the memory word length; the constant C2 is the pipelining factor; and the constant C is the number of memory references made by the housekeeping operations such as updating the relation table. Let $m=n$, $Z=K+L$ and $U=T/(C1*C2)$; we can rewrite the function as:

$f = U*n*n + Z*n + C$

Clearly the algorithm has the worst case asymptotical time complexity behaviour of $O(n^2)$. We can expect some contribution from pipelining and the speed of the complement operation. Of course we are not copying $n*m$ memory location but,

$(m*n)/C1$

memory locations.

Now we have to define the algorithm for the table representation. The complexity and the high cost of that algorithm should be apparent to the reader at this point. One relatively efficient way is to use the SCHT mechanism. The algorithm is as follows:

1.   Start from the beginning of the relation's table and proceed down the table. For each table record found, extract the right individual, hash to the SCHT with that individual, and establish its record. Link the right individuals' records in the SCHT by their TASE links as they are created.

(* Step 1 effectively creates the RIM set of the relation in the SCHT *)

2.   Perform the disconnection operation on the SCHT. Mark the beginning of the RIM set of the relation.

3.   Start from the beginning of the relation's table and proceed down in the table, record by record. For each record found extract the left individual, hash into the SCHT with that individual, and create a record of that individual (if there is no record for that individual in the SCHT). Make a separate copy of the RIM set, and set the TASE link of the left individual's record created to the copy of the RIM set. (* In step 3 we have established the universal relation of the given relation in SCHT, in an adjacency-list-like representation. *)

4.   Start from the beginning of the relation's table and proceed down in the table record by record. For each record found extract the left individual and extract the right individual, hash into the SCHT with the left individual, find the SCHT record of that individual, search the right individual's record in the bucket (copy of the RIM set) by following the TASE links between the records in the bucket, and delete it from the bucket.   (* In step 4 we have established the complement of the given relation in SCHT, in an adjacency-list-like representation. *)

5. Start from the beginning of the relation's table and proceed down in the table record by record. For each record found extract the left individual and hash with that individual into the SCHT. If a record of that individual is found and the TASE link field of that record does not contain the value "nil", create a new table record of the new relation, and copy the PML field of the left individuals record into the "left" field of the table record created. Follow the pointer found in the TASE link of the left individual's SCHT record and find the right individual's record (the record of the right individual that is in relation with the left individual in question under the new relation). Copy the PML field of that record into the "right" field of the table record created. If there remains other right individuals' records in the bucket, create a new table record for each of them and copy the PML field of the left individual's record into the "left" field and the PML field of the right individual's record (in turn) into the "right" field of that record. Link the table records created in this manner to each other by their "link" fields. Delete a left individual's record and the bucket of records connected to it when all the table records that can be created from them are created. (* Step 5 establishes the table of the complement of the given relation *)

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We have to define what would be the worst case for this algorithm. In the worst case, the relation may be a universal relation on its LEM set and RIM set. In that case the resulting relation is the empty relation. In that case step 4 of the algorithm becomes very costly. The worst case time complexity function is given below:

$$f = K*p + L*n + M*((p-m)+m*n) + \frac{N*m*n*(n+1)}{2} + Q*p + C$$

where:

p = n*m = relation size.

n = the cardinality of the RIM set of relation.

m = the cardinality of the LEM set of relation.

The first term corresponds to step 1, the second term corresponds to step 2, the third term corresponds to step 3, the fourth term corresponds to step 4, and the fifth term corresponds to step 5 of the algorithm. Constants K, L, M, N and Q represent the number of memory references made in each iteration of the corresponding steps. Constant C represents the number of memory references made by the housekeeping operations.

The third and fourth terms of the complexity function may not be clear to the reader, so we will explain how we found those terms. In step 3 we got through the entire table of the relation, but we made m separate copies of the RIM set of

the relation, where m is the number of distinct left individuals in the left column of the table. Hence we make (p-m) memory references without making the separate copy of the RIM set in the SCHT, for continuing to proceed down in the table. In addition to that we make m memory references and as a result of each of them we make a separate copy of the RIM set of the relation which requires n memory references. Note that even though we say that we make (p-m) memory references or m memory references, these are not the actual memory references we make. In fact these are the iteration factors to be multiplied by the constant number of memory references made in each iteration, which is represented as the averaged constant K. In the fourth step again we get through the table of the relation. Because in the worst case the size of the table is equal to:

$n*m$

we make n memory references for each of the distinct m left individuals. For each of those n memory references we have to search in the bucket one of the n right individuals, but after searching and finding one of the right individuals we delete the record of that individual from the particular bucket (that belongs to the one of the m left individuals) in question so the bucket size decreases by 1. A subsequent search for one of the remaining right individuals has to be done in a bucket smaller than the first bucket. By assuming

that in the worst case the bucket is searched to the end and the record of the right individual in question is always found as the last record of the bucket, we make:

n, (n-1), (n-2), (n-3), ..., (n-n+1)

memory references for each distinct left individual (that has a bucket in SCHT). We can write the above sequence in a compact form as:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

We multiply the iteration factors with the constant number of memory references made in each interation (which is in this case "N") to obtain the fourth term.

We accepted the relation size as the product of the cardinalities of the RIM and the LEM sets. In addition to that:

Let,

$U = N/2$

$Z = K + Q + M + U$

$S = L - M$

and of course $p = n*n$; then the complexity function becomes:

$f = U*(n^3) + Z*(n^2) + S*n + C$

a polynomial of degree 3.

So we conclude that the worst case asymptotical time complexity behaviour of this algorithm is $O(n^3)$. So the complexity function of this algorithm asymptotically dominates the one we defined for the previous algorithm.

Size Operation (#:C) :

This operation is defined only for sets. We often need the size or in other words, the cardinality of a set. This operation provides us with the cardinality of a given set

The algorithm is as follows:

1. Get the identifier of the set.

2. Hash with this identifier to the set table: find the record of the set.

3. Look up the "size" field of that record. If it is not negative then return the contents of that field, else do step 4.

4. Follow the PSS field of the record found in step 2: find the first set record. Proceed in the set structure by following the TASE link fields of the set records. Keep a count which is initialized to 0 and increment it for each record found above.

5. After the linked list structure of the set is exhausted, return the value of the count and also establish it in the "size" field of the set's record in the set table.

Clearly the algorithm goes through the set structure once and has the complexity function:

$$f = K*n + C$$

where constant K is the number of memory references made for
each record of the set found, constant C is the number of
memory references made in steps 1, 2, 3 and 5, and variable
"n" is the cardinality of the given set.

So the algorithm has the worst case time complexity
behaviour of $O(n)$.

Pair Operation (,) x,y = (x,y) :

That operation takes two individuals and constructs a
relation that has only one tuple in it. The first argument
individual becomes the left member and the second argument
individual becomes the right member of the unique tuple of
the resulting relation. The algorithms for Hash-Incidence-
Vector representation and table representation are equally
simple.

The algorithm for Hash-Incidence-Vector representation is
given below:

1. Get the argument individuals.

2. Hash to the relation table under the relation
identifier,

"(first argument individual)',`(second argument individual)"
establish its record in the relation table.

3. Hash to the LHT with the first argument individual
(of course after concatenating it with the above relation
identifier). Establish its LHT record, put 1 into the index

267

field of that record, put nil into the TASE field, and put the pointer to the memory location where the individual is actually stored into the PML field of that record. Establish the above relation identifier in the "Rid" field of that record. Put the pointer to that record into the PFLM field of the relation's record established in step 2.

4.   Repeat step 3 for the second argument on the RHT (i.e., hash into the RHT instead of LHT).

5.   Allocate a memory location for the incidence vector; set first bit from the left to 1 and the others to 0.

6.   Put the address of that memory location into the base field of the relation's record in the relation table.

7.   Put 1 into both the "|RIM|" and "|LEM|" fields of the relation's record.

The complexity function of the algorithm is:

$$f = C1$$

where:

constant "C1" is the number of memory references made in the algorithm. The algorithm, under every condition makes C1 memory references. So the algorithm has the worst (also the average) case time complexity behaviour of O(c).

The algorithm for table representation seems less costly than this algorithm, but we have to remember that we did not define the environment in which the table representation is defined.   That algorithm may also be as costly as the

268

previous algorithm depending on the environmental requirements.

The algorithm for table representation is given below:

1. Get the argument individuals.

2. Allocate a table record.

3. Put all into the "link" field of that record.

4. Put the pointer to the memory location where the first argument individual is saved into the "left" field, and pointer to the memory location where the second argument individual is saved into the "right" field of that table record.

This algorithm, like the previous algorithm, has a constant complexity function. But the constants are different. We conclude that both algorithms are cheap constant time algorithms.

Left Members (lem:R) :

This operation takes a relation identifier and returns the set of left members of that relation. A left member of a relation can be defined as the member which occurs on the left side of at least one tuple of the relation.

In our hash-incidence-vector representation this set is already available in the LHT as a collection of LHT records linked to each other by their TASE links. The header of that structure is the record of the relation in the relation table. The PFLM field of that record points at the first

left member of the relation which is the beginning record of the LEM set structure that we look for.

Even though we have this set readily available we have to carry it into the SHT in order to make the resulting set known by the system. We write the algorithm for this operation as follows:

1. Hash into the relation table with the identifier of the relation in question and find the record of the relation.

2. Follow the pointer found in the PFLM field of this record and the first record of the argument relation's LEM set.

3. Proceed in the LEM set record by record. For each record found, hash into the SHT with the individual being represented by this record and establish its SHT record in the SHT under the set identifier "lem:R". Link the SHT records created in this manner to each other as they are created. Keep a count beginning with 0 and increment it for each SHT record created.

4. Hash to the set table with the set identifier "lem:R"; establish the record of this set in the set table. Put the pointer to the first record of the resulting set structure (which is established in SHT) into the PSS field and the last value of the count into the CARD field of this record.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

We can write the time complexity function of this algorithm as follows:

$f = K*n + C$

where the constant K represents the constant number of memory references made for each record of the argument relation's LEM set in step 3. The constant C represents the constant number of memory references made in steps 1, 2 and 4. The variable n is the cardinality of the argument relation's LEM set.

So as can be seen, the algorithm has the worst case asymptotical time complexity behaviour of $O(n)$.

The algorithm that works on the table representation makes use of the SCHT mechanism for recognizing and removing duplicates of the individuals from the left column of the table. In fact if there was not the possibility of the duplicates in the left column, we could strip off the left column of the table without using the SCHT mechanism in order to obtain the LEM set of the relation. But even in this case the algorithm is costly and has the same asymptotical complexity behaviour as the algorithm that we will now define. The algorithm is as follows:

1. Start from the beginning of the relation's table; proceed down in the table record by record, by following the

271

links between the records. For each record found in this manner, extract the left individual, hash with that individual into the SCHT and if there is no record present for that individual in the SCHT, establish the record of that individual in the SCHT. If this record was the first record created mark it with pointer "P". Link the records created in the SCHT, in the above manner, to each other as they are created. Keep a count by beginning with 0 and increment it for each record created in the SCHT. Continue to examine the table records of the relation until the end of the relation is encountered.

2. Start from the beginning of the set created in the SCHT by following the pointer "P" which is set in step 1. For each set record found by proceeding in the set record by record, hash into the SCHT with the individual being represented by that record and find the SCHT entry to which the record in question is connected directly or indirectly (i.e., by being in a bucket which is connected to that hash table entry). Put nil into the hash table entry found, and put nil into the collision link fields of the records of the bucket if there exists a bucket which was connected to this hash table entry.

3. Establish a record of the resulting set in the set table under the set identifier:

"lem:"(relation's identifier).

Put the pointer "P" into the PSS field of that record, and put the last value of the count into the CARD field of that record.

We write the worst case time complexity function of that algorithm as follows:

f = K*p + L*n + C

where:

P = The size of the relation.

n = The cardinality of the LEM set of the relation.

K = Constant number of memory references made for each table record found in step 1.

L = Constant number of memory references made for each set record found in step 2.

C = Constant number of memory references made in step 3 of the algorithm.

By assuming that the LEM set and the RIM set of the relation have the common cardinality "n", in the worst case (i.e., when p = n*n) we can rewrite the worst case time complexity function as:

f = K*(n^2) + L*n + C

By looking at the exponent of the term with the larger exponent in the above function, we conclude that the worst case asymptotical time complexity behaviour of that algorithm is $O(n^2)$.

Members (mem:R) :

Members of a relation are the individuals which are either in the LEM set or in the RIM set of the relation or both. So this operation takes the LEM set and the RIM set of the relation and obtains the union of these sets. So the algorithm for hash-incidence-vector representation can be written as:

1.   Call the operation "Lem" with the relation being the argument.

2.   Call the operation "Rim" with the relation being the argument.

3.   Call the operation "Set Union" with the arguments being the identifiers of the LEM set and the RIM set of the relation which are:

"lem:"(relation's identifier).

"rim:"(relation's identifier).

4.   Establish the record of the new set in the set table under the set identifier:

"mem:"(relation identifier)

instead of the identifier created by the "Set Union" operation automatically. So if we say the complexity function of the "Set Union" operation is "F", the complexity function of the Members algorithm can be written as:

274

f = F + fl + f2

where:

    fl = The complexity function of the "Lem" operation.

    f2 = The complexity function of the "Rim" operation.

Since the complexity functions, F, fl, and f2 are all linear
functions the sum of those functions will also be a linear
function. Thus we conclude that the worst case asymptotical
time complexity behaviour of this algorithm is O(n), where
"n" is assumed to be the common cardinality of the LEM set
and the RIM set of the relation. We do the same in the table
representation case, but let's define the algorithm for the
table representation because the situation slightly differs
from the previous case. The algorithm for the table
representation is as follows:

    1. Start from the beginning of the relation's table.
Proceed down in the table record by record, by following the
links between the table records. For each record found in
this manner do the steps below:

        a. Extract the "right" individual, hash into the SHT
with that individual under the new set identifier "mem:R" and
create its record in the case there is no record for that
individual in the SHT already.

        b. Extract the "left" individual of that record, hash
into the SHT with that individual under the new relation's
identifier and create a record of that individual in the SHT

in the case that individual is not being represented by a record in the SHT already. Link the records created in this manner to each other by their TASE links as they are created. Keep a count beginning with 0 and increment it for each SHT (set) record created. Mark the first SHT record created in the above manner with pointer "P".

2. Do step 2 until the end of the relation's table is encountered.

3. Establish the record of the resulting set in the set table under the identifier:

"mem:"(relation's identifier).

Put pointer "P" into the PSS field of that record and put the last value of the count into the CARD field of that record.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The worst case time complexity function of that algorithm can be written as:

$f = K*p + C$

where:

p = Relation size/Table size.

K = Constant number of memory references made for each table record found in the steps 2 and 3, which is guaranteed to be greater than or equal to 4.

C = Constant number of memory references made by the housekeeping operations.

276

We know that in the worst case the relation may be equal to the cartesian product of its LZM set and RIM set, so by assuming LZM and RIM sets of the relation have the common cardinality "n" we substitute "n^2" in place of "p" in the above function, so the above function becomes:

$$f = K*(n^2) + C$$

So by looking at the exponent of the term with the larger exponent in the above function we conclude that the algorithm has the worst case asymptotical time complexity behaviour of $O(n^2)$.

Relation Intersection (R&S) :

This operation takes the intersection of the two relations. The result is a relation that contains those tuples which are both in the first argument relation and in the second argument relation.

Our original intend was to take the intersection of the incidence vectors of the operand relations to obtain the resulting relation's incidence vector. But the nature of the relational language turned out to be very dynamic; the left and right members of those relations may be in an arbitrary order depending on the order in which their records have been created. As a result of that, the LET and RET records take on different incidence vector indices and the resulting incidence vectors may have different structures, i.e., the 1's in the corresponding positions may mean different things

in the incidence vectors. In addition to that, in our system the union, intersection and difference operations on relations are defined even though the argument relations do not have the same LEM sets and the same RIM sets. This again does not let us use the union, intersection and difference of the incidence vectors of the operand relations to obtain the incidence vector of the resulting relation. One way to make use of the incidence vectors as intended originally is to enforce these restrictions in the definition of the language:

Let R and S be the operand relations:

1. R and S must have exactly the same LEM sets and the same RIM sets.

2. The set elements must be in order and should always be maintained in that order.

If we impose those restrictions on operand relations, it is guaranteed that the 1's in the corresponding positions of the incidence vectors mean the same thing, and then it becomes possible to utilize fast logical operations and pipelining on the incidence vectors. But the asymptotical behaviour of the algorithms remains the same, because the size of the incidence vector is:

$(n^2)/Cl$

where "n" is defined to be the cardinality of both LEM and the RIM set of the relation and "Cl" is the memory word length. We can decrease the cost by another constant factor

which comes from pipelining. So the complexity function of the algorithm becomes roughly:

$f = (n*n)/(C1*C2) + C$

where C2 is the pipelining factor.

In the present case, the language does not have the restrictions explained above and our algorithms will be defined according to the present definition of the language. We will see that worst case asymptotical time complexity behaviour of the algorithms will not change but the algorithms will be slightly inefficient.

We can express the intersection of two relations as given below:

$R\&S = P = \{ <x,y> \mid <x,y> \ \varepsilon\ R\ and\ <x,y>\ \varepsilon\ S \}$

Thus our algorithms are supposed to produce the relation P that satisfies the above condition, given the relations R and S as arguments.

The algorithm for Hash-Incidence-Vector representation is given below:

Let R be the first operand relation and S be the second operand relation.

1. Get the operand relations' identifiers.

2. Hash with the first and the second argument relations' identifiers to the relation table (RT), follow the pointers found in the PFLM and PFRM fields of the relation

R's record, and find the records of the first left member and the first right member of the relation R.

a. Start from the beginning of the LEM set of the relation R. For each record found by following the TASE links of the LEM records, until the LEM set is exhausted, hash into the LHT with the individual in question under the relation S. Check if a record of that individual is present in the LEM set of relation S, if so hash into the LHT under the new relation's identifier (which is: "R&S"). Establish a copy of that individual's record. Link the records copied in this manner to each other as they are created by their TASE links.

b. Repeat step 2-a for the RIM set of the relation R on the RHT, by also looking up the RIM set of the relation S. Keep a count for the new RIM set being created and increment it for each individual detected to be in the set. Put the updated value of that count into the index field of the record that represents the individual which has been detected to be in the set.

c. Start from the beginning of the new LEM set and establish the records' index fields which are connected to each other by their TASE link fields by beginning from 1 and incrementing the index by the last value of the count maintained in step 2-b. Furnish the index fields of these records.

d. Establish the record of the new relation in the
relation table under the new relation identifier. Establish
the cardinalities obtained by keeping count during the
creation of new LEM set and new RIM set into the "|LEM|" and
"|RIM|" fields of that relation record respectively.
Establish the pointers to the first records of the new LEM
and RIM sets into the PFLM and PFRM fields of that record.
Allocate a block of memory of size (|LEM|*|RIM|)/C, where C
is the memory word length. Put the beginning address of that
block into the base field of the relation's record.

3. Proceed in the LEM set of relation R, record by
record by following the TASE link fields of records. For
each record found, extract the individual being represented
by that record and do the steps below.

a. By starting from the beginning of the RIM set of
the relation R, proceed down in the RIM set record by record,
by following the TASE links between the records. For each
record found in this manner extract the individual being
represented by that record. Check the incidence vector
location corresponding to the tuple found in step 3 and step
3-b to see if it contains 1. If not, do nothing, else
continue with the steps below.

b. For each pair of individuals found in step 3 and
step 3-a, hash into the LHT with the individual found in step
3 and hash into the RHT with the individual found in step 3-b

281

under the relation S and check if this tuple is also present in the relation S (by using the reference algorithm).

      c. If so hash into the LHT and RHT with those individuals but this time under the new relation identifier. Using the reference algorithm, find the new incidence vector location that corresponds to this tuple and set that bit to 1.

      d. Else do nothing.

Note that the R&S and S&R refers to the same relation and when we create a relation with identifier "R&S" and establish relation's record in the relation table with that identifier a subsequent reference to the S&R may cause the same relation to be reconstructed redundantly. In order to eliminate this possibility we will accept a convention and design the system so that whenever R&S or S&R is referenced, we first look in the RT by hashing with the identifier "R&S". If no record is present, then we hash with identifier "S&R". If a record is present, we assume the original reference is S&R instead of R&S. From that point on the relation S&R participates in operations instead of R&S vice versa.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

Note that the given algorithm reflects the worst case, which means neither R&S nor S&R has been constructed

previously. The worst case complexity function of this costly algorithm can be written as:

$$f = K*m + L*n + Y*t + T*m*n + C$$

where first term corresponds to step 2-a, second term corresponds to step 2-b, third term corresponds to step 2-c, fourth term corresponds to step 3, "m" is the cardinality of the LEM set of the relation R, "n" is the cardinality of the RIM set of the relation R, "t" is the cardinality of the LEM set of the resulting relation. Constant C represents the number of memory references made in the steps other than the steps indicated above. Constant K represents the number of memory references made for each LEM set record in step 2-a. Constant L represents the number of memory references made for each RIM set record in step 2-b. Constant Y represents the number of memory references made for each LEM set record of the resulting relation in step 2-c. Constant T represents the number of memory references made for each pair of individuals found in step 3. Now let $m=n=t$ and $Z=(K+L+Y)$, the complexity function becomes:

$$f = T*n*n + Z*n + C$$

So the complexity function can be viewed as a second degree polynomial. We conclude that the algorithm has the worst case asymptotical time complexity behaviour of $O(n^2)$.

Now we have to consider how we could perform this operation in the case of the table representation. The algorithm is given below:

1. Take the relation with smaller size and make a separate copy of it.

2. Start from the beginning of the second relation's table and proceed down in the table. For each record found, extract the left individual and search for that individual in the left column of the new table. If it is found compare the right individuals of the current record of the new table and the record of the second relation above. If a common tuple is found in this manner, extract the new table's record representing that individual from its place and carry it to the top of the table. If it is the first record carried to the top, mark it with pointer Z.

3. Repeat step 2 until the table of the second relation is exhausted or until the pointer Z points at the bottom of the new table.

4. Delete the records below the Z (if there are any).

In the best case two operand relations may almost be the same. By factoring out the constants, the complexity function can be written as:

$$f = p + (p-1) + (p-2) + \ldots\ldots\ldots + (p - p + 1)$$

where "p" refers to the size of both the operand relations. This can be rewritten as:

284

$$f = \sum_{k=1}^{P} (p-p+k) = \sum_{k=1}^{P} k = p*(p+1)/2$$

So we know that we can not do better than that. Now we have to decide about the worst case. In the worst case the operand relations may be disjoint, in which case for each tuple of the first relation we go through exhaustively the whole copied relation and we delete the copied relation as a whole at the end. So the worst case complexity function becomes:

$f = K*p*r + L*r + M*r + C$

where Constant K is the number of memory references made in each iteration of step 2, constant L is the number of memory references made while copying each record of the second relation in step 1, constant M is the number of memory references made while deleting each record of the copied relation in step 4, constant C is the number of memory references made by the housekeeping operations, variables p and r are the sizes of the operand relations.

We know that in the worst case both operand relations may be universal relations on their LEM and RIM sets. In that case, as explained before, the sizes of the relations are the product of the cardinalities of their LEM and RIM sets respectively. Let n be the common size of the LEM and RIM sets of both relations and let the operand relations be the universal relations on their LEM and RIM sets or in other words,

285

let the operand relations' sizes be equal to the cartesian product of their LEM and RIM sets, which in turn means:

p = q

The complexity function becomes:

$$f = K*(n^4) + F*(n^2) + C$$

where F = L + M

We conclude that the algorithm has the worst case asymptotical time complexity behaviour of $O(n^4)$, which is a very costly algorithm. Again there exists many efficient algorithms to do this operation but they are not compatible with the dynamics of our system, and they have other redundancies when they are combined with the whole system. For example, if we maintain our tables representing relations in sorted order this algorithm may be made simpler and cheaper by using one of the fast searching algorithms, but maintaining the tables in sorted order is a significant burden in such a dynamic system, because we may recompute the individuals in the relation and we may add new tuples as a result of the relational operations, and so on. These operations are so frequent that every time sorting the tables is a significant burden.

Relation Union (R|S) :

This operation takes two relations and produces a relation in which each tuple is either present in one operand relation or in the other. That can be stated formally as:

R|S = { <x,y> | <x,y> ε R OR <x,y> ε S }

The set of tuples such that each tuple is either in R or in S.

The algorithm for Hash-Incidence-Vector representation is as given below:

Let the first operand relation be R and the second operand relation be S.

1. Find the records of the relations R and S in the relation table.

2. Follow the PFLM and PFRM fields of the relation R's record, find the first left member's record and the first right member's record in the LHT and in the RHT respectively.

3. Repeat step 2 for relation S.

4. Copy the RIM set of relation R in the RHT under the relation identifier "R|S" (as it was done in many previous algorithms). Copy the RIM set of relation S under the relation identifier "R|S". As the records are created during the copying operation, establish the new indices in the index fields of the records by keeping an index count and incrementing it for each record created, then by putting the current value of it into the index field of the record created recently. Link the records created in the above manner to each other by their TASE link fields.

5. Repeat step 4 for the LEM sets of relations R and S, on the LHT. But this time, keep one count for establishing

287

indices and another count for finding out the cardinality of the new LEM set. While establishing the indices of the records, begin with 1 and increment the index count by the cardinality of RIM set for each record created, and put the updated value of the index count into the index field of the record recently created. For each record created increment the cardinality count.

6. Establish the new relation's record in the relation table under the identifier "R|S". Establish the LEM set cardinality count in the $|LEM|$ field and the RIM set index count in the $|RIM|$ field of that record. Put the pointers to the records of the first left member and the first right member into the PFLM and PFRM fields of that record respectively. Allocate a memory block as large as:

$(|LEM| * |RIM|)/C$

where C is the memory word length. Put the beginning address of that block into the base field of the above record.

7. Start from the beginning of the LEM set of relation R. For each record found by following the TASE links between the records until the LEM set is exhausted; do the steps below.

a. Start from the beginning of RIM set of relation R, follow the TASE links between the records, and proceed down in the RIM set, record by record.

288

b. For each tuple found which is being represented by the record pair found in step 7 and in step 7-a, reference the incidence vector of relation R (by using the reference algorithm). If the corresponding incidence vector entry is 1, hash with the left individual of the tuple into the LHT and with the right individual of the tuple to the RHT. Extract the indices of the corresponding records, then reference the incidence vector of the new relation by using the reference algorithm and put 1 into the incidence vector entry found.

8. Repeat step 7 for the relation S.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The algorithm seems to be expensive, especially steps 7 and 8 are the costly steps of this algorithm. We write the worst case complexity function of this algorithm as follows:

$$f = K*(m+p) + L*(n+q) + T*(m*n) + P*(p*q) + C$$

where:

m = cardinality of the LEM set of R.

n = cardinality of the RIM set of R.

p = cardinality of the LEM set of S.

and q = cardinality of the RIM set of S.

Constants in front of each term indicate the number of memory references made for each iteration of the corresponding step. The correspondence between the steps of the algorithm and the

terms of the function as follows: the first term corre-
sponds to step 4, the second term corresponds to step 5, the
third term corresponds to step 7, and the fourth term corre-
sponds to step 8 of the algorithm. Constant C is the number
of memory references made by the other steps of the algo-
rithm.

Let:

   $n = m = p = q$ and

   $R = 2*K + 2*L$ and

   $U = T + P.$

Then the complexity function becomes:

   $f = U*(n^2) + R*n + C$

So we conclude that the worst case asymptotical complexity
behaviour of the algorithm is $O(n^2)$.

   How could we perform the same operation on the table
representation? Again we have to get help from SCHT
mechanism in order to make the algorithm as efficient as we
can. The algorithm is as follows:

   Let R be the one operand relation and S be the other.

   1. Start from the beginning of the table of relation R,
proceed down in the left column of the table, record by
record, for each left individual found in that way, hash with
that individual into the SCHT, create a record of that
individual in SCHT and connect it directly to the hash table
entry found. If it is found out that a record of a left

individual is already connected directly to this hash table entry (i.e., if the collision occurs), search for the next empty hash table entry in the hash table, and connect the record of the individual in question to that hash table entry directly. If a record of that individual is already established in the SCHT previously, do nothing. Create an SCHT record of the right individual of the current tuple, connect it to the record of the left individual created above. If there is a bucket of records connected to the left individual's record (as a result of establishment of previous tuples), add the record of right individual in question to the end of that bucket. For each hash table entry found and used up in this manner, set a pointer to that hash table entry and put that pointer into the temporary array of type pointer.

2. Repeat step 1 for relation S; establish the records of the right individuals in the buckets connected to the records of the left individuals if they are not already present in the buckets (i.e., do not allow repetition of same right individual's record in the same bucket). (* In the above steps, we handled the collisions by rehashing and we use bucketing to relate the right individuals with the left individuals. In fact we created another representation of the resulting relation in SCHT. As we know this representation technique is called an "Adjacency list". Now

291

the remaining steps of the algorithm are to convert that representation into our "table" representation. *)

3. Start from the beginning of the temporary pointer array and find each occupied hash table entry in turn. For each bucket found connected to this hash table entry, do these steps:

a. Extract the first record from the bucket, create a table record and put the PML field of the record extracted from the bucket into the "left" field of that table record.

b. Extract the next record from the bucket, and copy the PML field of that record into the "right" field of the table record.

c. While there remains a record in the bucket, create a new table record, copy the PML field of the record found in step 3-a into the "left" field, and the PML field of the remaining record into the "right" field of the table record created.

4. Link the table records created in step 3 to each other by their TASE link fields as they are created.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

In the worst case each relation may be universal relations on their LEM and RIM sets. In addition to that, the LEM sets of the argument relations may be the same, but the RIM sets may be disjoint. This also implies that the

292

relations are disjoint. Under these considerations, in step 2 the algorithm searches the entire bucket and can not find the record of the right individual in question. It then adds the record of that individual to the end of the bucket. We have to note that in the worst case indicated above each bucket has as many records as the cardinality of the RIM set of the relation R. Because the LEM sets of the relations are the same, each bucket constructed in SCHT is searched in the manner explained above. In addition to that, after termination of steps 1 and 2 the resulting bucket sizes are the total of the cardinalities of the RIM sets that belong to the relations R and S.

The worst case complexity function of this algorithm can be written as:

$$f = K*m*n + L*(p*q*n) + T*m*(n+q) + C$$

where:

m = cardinality of the LEM set of relation R.

n = cardinality of the RIM set of relation R.

p = cardinality of the LEM set of relation S = m

q = cardinality of the RIM set of relation S

In the above function:

first term corresponds to step 1.

second term corresponds to step 2.

third term corresponds to step 3 of the algorithm.

$m = n = p = q$ and $Z = 2*T + K$, the complexity function becomes:

$$f = L*(n^3) + Z*(n^2) + C$$

So we conclude that the worst case asymptotical time complexity of that algorithm is $O(n^3)$.

Relation Difference (R - S) :

This operation takes two relation identifiers as argument and produces another relation which has only those tuples that are in the first operand relation and not in the second operand relation. This can be formally stated as:

$$R - S = \{ <x,y> \mid <x,y> \text{ e } R \text{ and } not<x,y>e \text{ } S \}$$

Algorithm for Hash-Incidence-Vector representation is given below:

1. Get the relations' identifiers and record the identifier of the first argument relation as reference.

2. Find the records of the relations by hashing with their identifiers to the Relation Table.

3. Follow the pointers in the PFLM and PFRM fields of the reference relation's record; find the first LEM and first RIM records of that relation.

4. Proceed in the LEM set of the relation; for each record found, do the steps below:

   a. By starting from the first record in the RIM set of the reference relation, proceed down in the RIM set.

294

b. For each tuple found being represented by the records found in steps 4 and 4-a, hash with the left individual of the tuple into the LHT and with the right individual to the RHT under the second relation's identifier. Test if that tuple is already present in the second relation or not (by using the reference algorithm).

c. If so do nothing.

d. Else hash with the left individual in question into the LHT and with the right individual in question to the RHT, under the new relation's identifier, which is "R-S", where R is the identifier of the reference relation and S is the identifier of the other operand relation. Establish the LHT and RHT records of that individual in the LHT and in the RHT respectively. If they are the first records established in this manner set pointer P to the left individual's record (in the LHT) and set pointer to the right individual's record (in the RHT). Link the records created in this manner to each other as they are created.

e. Keep a RIM set index count to furnish the index fields of the RIM records and update it as the records are created. Keep LEM set cardinality count for finding out the cardinality of the resulting LEM set; increment it when each LEM record is created.

5. Repeat step 4 until the LEM set of reference relation is exhausted.

6. Hash to the relation table with the new relation's identifier ("R-S"), establish its record, put pointer P into the PFLM field, pointer Q into PFRM field, the last value of the LEM set cardinality count into the "|LEM|" field, and the last value of the RIM set index count into the "|RIM|" field of this record. Allocate a block of memory as large as:

$$(|LEM| * |RIM|)/C$$

Where C is the memory word length. Put the beginning address of that block into the base field of the relation's record.

7. Start from the beginning of the LEM set of the new relation, proceed down in the LEM set of new relation record by record. By beginning with 1 and incrementing the index every time by the last value of the RIM index count, furnish the index fields of the LEM records. In addition, for each record found do the steps below:

a. By starting from the first record of the RIM set of the new relation proceed down in the RIM set record by record.

b. For each tuple found (being represented by the record pair found in steps 7 and 7-a) hash with the left individual into the LHT and with the right individual to the RHT under the reference relation and reference the incidence vector location corresponding to that tuple by using the "reference" algorithm. If a 1 is found in the corresponding entry, reference the second relation in the same manner to

296

check if this tuple is also present in the second relation. If so do nothing; else put 1 into the corresponding incidence vector entry of the new relation.

We could write an algorithm which is less costly as follows: It makes a separate copy of the incidence vector and the LEM and RIM sets of the reference relation, then it goes through the second relation and deletes the entries from the copied incidence vector that corresponds to the tuples found in the second relation. But suppose that the reference relation is a very large relation and the intersection of two operand relations is also very large so that the result of the operation is a relation that has only a few tuples. As can be seen, we are allocating an incidence vector as large as the reference relation's incidence vector for a few tuples; we are wasting a lot of storage. On the other hand our previous algorithm uses an amount of memory as large as needed.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

Clearly the algorithm is an expensive algorithm, the worst case complexity function can be written as:

$f = K*(m * n) + L*(p * q) + C$

Where the first term corresponds to step 4, and the second term corresponds to step 7 of the algorithm. Constant C is the number of memory references made by the other steps of

the algorithm. Variable m is the cardinality of the LEM set of relation R, variable n is the cardinality of RIM set of relation R, variable p is the cardinality of the LEM set of the resulting relation, variable q is the cardinality of RIM set of resulting relation. Let m=n=p=q and T = K+L, then the complexity function becomes:

f = T*n*n + C

So we conclude that the algorithm has the worst case asymptotical time complexity behaviour of $O(n^2)$.

The algorithm for table representation is a costly algorithm, as it was before in the other operations. The dynamics of the system causes the individuals to be out of order in the left and the right column of the table. This, in turn, causes exhaustive searches in the columns of the table. Even though the deficiency is obvious, we might wonder how inefficient the algorithm is relative to the previous algorithm. The algorithm is given below:

1. Start from the beginning of the table of the reference relation. Make a separate copy of that table.

2. Start from the beginning of the other operand relation's table. For each tuple found by proceeding through the table, record by record, search in the copy of the reference relation for that tuple; if it is present then delete it from the copy of the reference relation's table.

3. Repeat step 2 until the relation (other than the reference relation) is exhausted.

In the worst case the relation R and the relation S may be disjoint (i.e., R-S = null relation). That means for every tuple found in step 2 we go through the copy of the reference relation exhaustively (because there is no tuple in common the search is unsuccessful each time). Under this circumstance the worst case complexity function can be written as:

$$f = K*p + L*r*p + C$$

The first term corresponds to step 1, and the second term corresponds to steps 2 and 3, of the algorithm.

In the above function, "p" is the size of the reference relation, "r" is the size of the other operand relation and constant C is the number of memory references made by the housekeeping operations (such as updating relation table, etc.). Constants K and L represents the constant number of memory references made at each iteration of step 1 and step 2, which are expected to be small.

Let the cardinality of the LEM sets and the RIM sets of the relations be the same and equal to "n". Let the relations be the universal relations on their "MEM" set (MEM = LEM | RIM). That means the size of the relations are the same under these assumptions. The worst case complexity function becomes:

$$f = K*(n^2) + L*(n^4) + C$$

So the algorithm has the worst case asymptotical time complexity behaviour of $O(n^4)$. Assuming the absence of the constant factors, 160,000 memory references are necessary when n=20. On this basis we may say that the algorithm is practically inexecutable. We have to remember from previous discussions that keeping the tables in sorted order is not a solution in the present definition of the system.

Restriction Operation (S/R\S) :

It is sometimes useful to restrict both the domain and the codomain of a relation. The restriction operation, given an argument set and the relation identifier, restricts the RIM set and the LEM set of that relation to the given set. That means the RIM set and the LEM set of the relation can contain only those individuals that are in the argument set. We can state it more carefully as follows:

$$S/R\backslash S = \{ <y,x> |\ yRx \wedge y\epsilon S \wedge x\epsilon S \}$$

The algorithm for Hash-Incidence-Vector representation is as follows:

1. Get the relation identifier and the argument set identifier and hash with those identifiers to the relation table and the set table respectively. Find the records of the first right member and the first left member of the relation by following the pointers found in the PFRM and the PFLM fields of the relation's record respectively.

2. If the argument set is extensionally represented (i.e., the PSS field is not nil), start from the beginning of the linked list structure of the RIM set, proceed down in that linked list. For each RIM set record found by following the TASE links between the records, hash with the individual being represented by that record into the SHT, under the argument set's identifier, to test if there exists an SHT record for that individual in the SHT. If there exists a record for that individual in the (SHT) argument set, copy that record in the RHT under the new relation identifier:

(set identifier)'/'(relation identifier)'\'(set identifier)

Link the RHT records to each other as they are created. Keep a RIM set cardinality count and increment it each time a RIM record is created by beginning with 0.

3. After the RIM set is exhausted do step 2 for the LEM set of the relation in the LHT.

4. If the argument set is being intensionally represented, begin from the beginning of linked structure of the LEM set in the LHT. Proceed down in the LEM set record by record by following the TASE links between the records and for each RIM set record found in this manner, test if the individual being represented by that record is a member of the argument set. (This membership test will be explained later in the discussion of the intensional representation techniques). If it is a member, copy that RHT record in the

301

RHT under the new relation identifier indicated above. Link the RHT records created in the RHT in the above manner to each other as they are created. After finishing with the LEM set of the relation, begin from the beginning of the RIM set of the relation and do the same as it was done for the LEM set, but this time for the RIM set of the relation in the RHT. Keep a LEM cardinality count and increment it for each LHT record created by beginning with 0; in the same manner keep a RIM cardinality count beginning with 0 and increment it for each RHT record created while performing the above functions.

5. Hash to the relation table under the new relation identifier indicated above and establish the record of the new relation in the relation table. Copy the BASE field of the original relation's record into the BASE filed of the new relation's record. Put the last value of the RIM cardinality count into the |RIM| field, and put the last value of the LEM cardinality count into the |LEM| field of the new relation's record. Put the pointers to the first records of the LEM and the RIM set of the original relation into the PFLM and PFRM fields of that record.

Now we will do the worst case asymptotical time complexity analysis for this algorithm.

We will assume that in the worst case the argument set is a super set of both the RIM set and the LEM set of the

302

original relation; in that case we necessarily copy the whole RIM set and the LEM set of the original relation in order to obtain the LEM set and the RIM set of the new relation. We write the worst case time complexity function of that algorithm as follows:

$f = L*m + S*n + C$

where:

n = The cardinality of the RIM set of the original relation.

m = The cardinality of the LEM set of the original relation.

L = The constant number of memory references made while copying each LEM set record.

S = The constant number of memory references made while copying each RIM set record.

In the above function the second term corresponds to step 2, the first term corresponds to step 3, and the last term corresponds to steps 1 and 5 of the algorithm.

Let the cardinalities of the LEM and RIM sets of the relation be equal and $T = L+S$; then the complexity function becomes;

$f = T*n + C$

So we conclude that the worst case asymptotical time complexity behaviour of this algorithm is $O(n)$ where "n" is

303

the common cardinality of the LEM and RIM sets of the relation.

Now we have to ask how could we do this operation on the table representation. The algorithm is as follows:

1. Start from the beginning of the linked list structure of the relation's table, and proceed down in that linked list record by record. For each record found in this manner, hash into the SHT with the individual being represented by the "right" field of that table record under the argument set's identifier. If there exists an SHT record for that individual, hash into the SHT with the individual being represented by the "left" field of that table record under the argument set's identifier. If there exists an SHT record for that individual also create a new table record, copy the "left" and the "right" fields of the original table record into the corresponding fields of the new table record. Link the new table records created in this manner to each other by their "link" fields as they are created.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

In the worst case the argument set may be a super set of the LEM set and the RIM set of the original relation. In that case we necessarily make a separate copy of the whole table of the original relation in order to obtain the table of the new relation. From that point on the worst case time

304

complexity analysis of the above algorithm is exactly the
same as the worst case time complexity analysis of the
algorithm for the "Right Restriction" operation that works on
the table representation. So the reader should refer to the
analysis done in the "Right Restriction" operation case.

Dual Application (R # S) :

This operation takes two relation identifiers and
constructs a new relation which relates the individuals with
the pairs. Given a pair in relation with an individual under
the resulting relation, the left individual of this pair is
the result of application of the first argument relation (R)
to this individual and the right individual of this pair is
the result of application of the second argument relation (S)
to this individual. So the resulting relation's right
members set consists of individuals and the left members set
consists of pairs.

The algorithm for Hash-Incidence-Vector representation is
as follows:

Let R be the first argument relation and S be the second
argument relation.

1. Find the records of the relations in the relation
table by hashing with the relation identifiers to the
relation table.

2. Follow the pointer in the PFRM field of the relation
R's record; find the first right member's record in the RHT.

305

3. Proceed in the RIM set of the relation R by following the TASE links between the records; for each RIM set record found in this manner do the steps below:

a. Hash with the individual being represented by the current RIM set record to the RHT under the relation S. If a record for that individual is also present in the RIM set of relation S. Hash into the RHT with the individual in question under the new relation identifier, "R#S". Establish a record for that individual; if this is the first record established in RHT for the new relation, mark it with pointer P.

b. Apply relation R to the individual in question by calling the "apply" algorithm; record the pointer returned in variable "temp1".

c. Apply relation S to the individual in question by calling the "apply" algorithm; record the pointer returned in pointer variable "temp2".

d. Follow the pointer recorded in variable "temp1" and find the individual resulting from the application of R to the current right individual. In the same manner, follow the pointer recorded in variable "temp2" and find the second individual resulting from the application of the relation S to the right individual in question.

e. Call algorithm "pair" with those individuals and record the pointer to the record of the "pair" relation resulting, in the variable "temp3".

f. Hash to the LHT under the new relation identifier "R#S", establish a LEM record of that relation in LHT, put the pointer recorded in pointer variable "temp3" into the PML field of that record. If it is the first LEM record created in this manner, mark it with pointer Q. Set the PRLM link of the current RIM set record to the current LEM set record created above. Link the records created in the LHT and RHT for the new relation to each other by their TASE links as they are created (except the "dummy" records). Keep a RIM index count and LEM cardinality count for each RIM set record created, increment the RIM index count and put the updated value of the RIM index count into the index field of the RIM set record created. For each LEM record created increment the LEM cardinality count. (* According to our convention of establishing indices in the LEM records we can not establish the indices in the LEM records until after the cardinality of the RIM set of the relation becomes evident. *)

4. Start from the beginning of LEM set of the relation by following the pointer Q, for each record found by following the TASE links between the records, increment the LEM index count by the last value of the RIM index count (by

beginning from 1) and put the updated value of the LEM index count into the index field of the record in question.

5.   Allocate a block of memory as large as:

(LEM-cardinality-count*RIM-index-count)/C

where C is the memory word length.

Initialize that vector to all zeros (* This may turn out to be the costly part of the algorithm if we are not using pipelining. *)

6.   Hash to the relation table with the new relation's identifier "R#S", establish its record and put the pointers P and Q into the PFRM and PFLM fields of that record respectively.   Put the beginning address of the incidence vector allocated in step 5 into the BASE field, put the LEM cardinality count into the |LEM| field, and put the RIM index count into the |RIM| field of that record respectively.

7.   Start from the beginning of the RIM set of new relation.   For each RIM set record found by following the TASE links between the records get the index of that record. Follow the pointer found in the PRLM field of this record and find the record of the left individual that is in relation with the current right individual in LHT. Get the index of that record, reference the incidence vector with those indices by calling the algorithm "reference"; and put 1 into the corresponding incidence vector entry.

308

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The RIM set of the resulting relation is in fact the intersection of the RIM sets of the argument relations. So in the worst case the cardinality of the RIM set of the resulting relation may be as large as the cardinality of the RIM set of the relation R. In addition to that, in the worst case each right individual of the resulting relation may correspond to a unique left individual. That means we create a pair relation for each right individual found to be in the RIM set of the resulting relation. That also means that the cardinality of the RIM set of the resulting relation is equal to the cardinality of the LEM set. Under these considerations, we write the worst case time complexity function as:

f = S*n + T*n + U*(n*n)/D + V*n + C

where n is the cardinality of the RIM set of relation R, constant D is the memory word length, constant C is the number of memory references made in steps 1, 2 and 6. In the above function: First term corresponds to step 3, second term corresponds to step 4, third term corresponds to step 5, fourth term corresponds to step 7 of the algorithm. In the third term the expression:

(n x n)/D

stands for the size of the incidence vector in terms of the
number of memory locations occupied. As we mentioned above,
in the worst case the cardinality of the RIM set and the
cardinality of the LEM set of the resulting relation are
equal to the cardinality of the RIM set of elation R (which
is "n").

Let:

$Z = S + T + V$,

and $W = U/D$

then the complexity function becomes:

$f = W*(n^2) + Z*n + C$

So we conclude that the worst case asymptotical time
complexity behaviour of this algorithm is $O(n^2)$.

Now we have to think about how the operation could be
performed on the table representation. The algorithm for the
table representation is simpler than the previous algorithm.
The algorithm is as follows:

Let R be the first argument relation and S be the second.

1. Start from the beginning of the relation S's table
and proceed down in the table record by record by following
the links between the table records. For each record found,
extract the right individual and hash with that individual
into the SCHT, establish a SCHT record for that individual in
SCHT, extract the left individual and create another SCHT
record for that individual, and link the record of the right

310

individual to that record by its TASE link. If after hashing
with that right individual, a SCHT record of that individual
is found to be present in SCHT, do nothing. Link the records
created in the SCHT to each other by their TASE links.
(* I.e., establish the SCHT record of a right individual and
the record of the corresponding left individual in SCHT only
once. *) (* As the result of the execution of step 1, each
right individual has a record in SCHT and is followed by the
record of the left individual which would have resulted from
application of relation S to that right individual. Note
that the SCHT record created for the left individual is not
connected to any SCHT entry. *)

2. Start from the beginning of relation R's record.
Proceed in the table of the relation R. For each table
record of R found in this manner, extract the right
individual, hash with that right individual into the SCHT.
If a record of that individual is not already present in SCHT
then do nothing, and continue with the next table record in
R. Otherwise create a new table record, put the pointer to
the right individual in question into the "right" field of
that record, and create another relation with only one table
record. Extract the pointer to the left individual from the
"left" field of the current table record in R, put that
pointer into the "left" field of the new relation's (pair)
record, follow the TASE link of the SCHT record of the right

individual in question and find the next SCHT record in the SCHT. Extract the pointer to the left individual (that was in the "left" field of the relation S's table record before) from that record and in the same manner put that pointer into the "right" field of the pair relation's record. Establish the record of the pair (a singleton relation) in the relation table. Put the pointer to this record into the "left" field of the new table record created for the resulting relation above. Delete the SCHT record for the right individual in question, and the following SCHT record (which belongs to the left individual that is in relation with the right individual in question under the relation S) from the SCHT. Update the TASE links between the SCHT records appropriately. (* Because there is no need to create a table record of the resulting relation again for that right individual. *) Link the resulting relation's records to each other as they are created by their link fields.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As we see the algorithm is simpler than the previous algorithm. The complexity function of that algorithm is as follows:

f = K*p1 + L*p2 + C

We know that in the worst case the sizes of the relations are equal to the product of the cardinalities of their LEM and

RIM sets. Since we have accepted the LEM set and/or the RIM set cardinality as a measure, we have to write the function in terms of these cardinalities. Before doing that we have to explain the meanings of the constants and variables in the above function. The variable "p1" is the size of the relation S, variable "p2" is the size of the relation R, the constant K is the number of memory references made for each iteration in step 1, and in the same sense, the constant L is the number of memory references made for each iteration in step 2. In the above function, the first term represents the step 1, and the second term represents the step 2 of the algorithm. The constant C is the number of memory references made by the housekeeping operations such as updating the relation table.

Let:

$p1 = p2 = n*n$

where "n" is the common cardinality of the LEM and RIM sets of the argument relations, and let:

$Z = K+L$

Then the complexity function becomes:

$f = Z*n*n + C$

So we conclude that the worst case asymptotical time complexity behaviour of that algorithm is $O(n^2)$.

313

Parallel Application (R||S) :

This operation takes two relation identifiers and constructs a new relation, in which each left member and each right member is a pair. Given a tuple of this relation, the left component is a pair, in which the left individual of the pair is the result of application of the first argument relation to the left individual of the right component (which is also a pair) of the tuple. Analogously, the right individual of the left member (which is a pair) is the result of application of the second argument relation to the right individual of the right component.

This algorithm is naturally more complex than the previous algorithms. The algorithm for Hash-Incidence-Vector representation is as follows:

Let the first argument relation be R, and the second argument relation be S.

1. Start from the beginning of RIM set of relation R and proceed down in the RIM set record by record. For each RIM set member found in this manner do the steps below:

a. Start from the beginning of RIM set of the relation S and find each record representing a RIM set individual in turn by following the TASE links between the records of the RIM set.

b. For each pair of individuals found in step 1 and step 1-a (i.e., we find the records that represent the

314

individuals then we extract the individuals by following the pointers in their PML fields), call the algorithm "pair" with the individual obtained in step 1 as the first argument individual and the individual obtained in step 1-a as the second argument individual. (* Note that the algorithm "pair" first looks in the relation table for that "pair" relation and executes this operation only, if this relation has not been created before. *)

c. Hash to the RHT with the identifier of the pair constructed by "pair" operation (say "(x,y)" under the relation identifier "R||S". Create a RHT record and put the pointer to the record of the pair relation established in relation table into the PML field of that record. Every time a RIM set record is created, put the updated value of the RIM set index count into the index field of that record. If the record is the first RIM record created in this manner mark it with pointer "Z".

d. Apply the relation R to the individual obtained in step 1 by calling the "Function application" algorithm. Call the individual returned by this algorithm W.

e. Apply the relation S to the individual obtained in step 1-a by calling the "Function application" algorithm. Call the individual returned Y.

f. Repeat step 1-b for the individuals "W" and "Y".

g. Repeat step 1-c for the pair "(W,Y)" on the LHT, but this time do not furnish the index fields of the records. Keep a LEM set cardinality count and increment it for each record created. Mark the beginning record of the LEM set with the pointer U. (* In this case duplications of the LEM records may occur. To prevent this, our algorithm does not create the LEM record of a pair if there is already a LEM record for that pair in the LEM set of the new relation. This is a property of the hashing mechanism. *) Set the PRLM link of the current RIM record of the new relation to the LEM record of the left individual constructed in steps 1-d through 1-g. (* The result of one iteration of step 1 yields the establishment of one right member and one left member of the new relation "R||S" that are in relation with each other under this relation. The result of exhaustive execution of step 1 is the creation of complete LEM and RIM sets of new relation. *)

2. Start from the beginning of the LEM set of new relation, and proceed down in the LEM set record by record by keeping an index count and incrementing it by the last value of the RIM set index count for each record passed. Put the updated count into the index field of each record passed.

3. Allocate a block of memory as large as:

(LEM-cardinality-count*RIM-index-count)/C

where C is the memory word length. Initialize that memory block to all zeros.

4. Hash to the relation table under the relation identifier R||S, create a new RT record, put pointers Z and U into the PFRM and PFLM fields of that record respectively, put the beginning address of the memory block allocated in step 3 into the BASE field, and put the LEM cardinality count and RIM index count into the |LEM| and |RIM| fields of that record respectively.

5. Start from the beginning of the RIM set of new relation and proceed down in the RIM set record by record. For each record found:

a. Extract the index field of the record.

b. Follow the pointer in the PRLM link field of that record and find the record of the left individual (pair) in relation with the current right individual. Extract the index field of that record.

c. Reference the new relation's incidence vector with those indices obtained in steps 5-a and 5-b and put 1 into the corresponding incidence vector entry.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

The complexity function of this algorithm can be written as follows:

$$f = N*n1*n2 + T*m1*m2 + (R*n1*n2*m1*m2)/D + U*n1*n2 + C$$

317

in which the first term corresponds to step 1, second term corresponds to step 2, third term corresponds to step 3, fourth term corresponds to step 5 of the algorithm. Constant C refers to the constant number of memory references made by the other steps of the algorithm. In the above function:

m1 = the cardinality of the LEM set of relation R,

n1 = the cardinality of the RIM set of relation R,

M2 = the cardinality of the LEM set of relation S,

n2 = the cardinality of the RIM set of relation S.

In the above function the second, third and fourth terms may not be clear to the reader. First of all we are creating records for all possible pairs that can be constructed from the RIM set individuals of the argument relations and each one of those pairs becomes a right individual of the new relation. Hence the cardinality of the RIM set of new relation is:

$|RIM| = n1*n2$ (n1 and n2 are as defined above)

This may not be true for the cardinality of LEM set of the new relation since two or more RIM set individuals of the new relation may be in relation with one LEM set individual. But in the worst case the cardinality of the LEM set becomes the product of the cardinalities of the LEM sets that belong to argument relations. That means:

$|LEM| = m1*m2$ (m1 and m2 are as defined above.)

318

We know that the size of the incidence vector is computed by the formula:

SIZE = (|LEM|*|RIM|)/D

where:

D = Memory word length

So the size of the incidence vector can be written in terms of the cardinalities of the LEM sets and the RIM sets of the argument relations as:

SIZE = (n1*n2*m1*m2)/D

So while we are establishing the indices of the new relation's LEM records in step 2, we make a number of memory references proportional to:

m1*m2

and while we are initializing the incidence vector we make a number of memory references proportional to:

(n1*n2*m1*m2)/D

In the same sense, while we are establishing the 1's in the new incidence vector we are making a number of memory references proportional to:

(n1*n2)

Now let n1=n2=m1=m2 = n, Z = (N+T+U) and H=R/D, then the complexity function of the algorithm can be written as:

$$f = H*(n^4) + Z*(n^2) + C$$

The algorithm is terribly expensive. The reason is we had to initialize the incidence vector of the resulting relation

319

which made the algorithm an order four algorithm. But even in the absence of this term (initialization), the algorithm is terribly expensive because the constant Z is expected to be so large. In addition to that time deficiency, the algorithm is storage inefficient. We have to construct the huge incidence vector of the resulting relation, because the relation may participate in subsequent operations. So the algorithm is practically infeasible. Fortunately the intensional algorithms for this operation are cheap so we can do this operation intensionally.

Now we have to define the algorithm for the table representation. In this algorithm we will use two scratch hash tables, SCHT1 and SCHT2, to make the algorithm easier to understand. Of course the algorithm can be defined by using only one scratch hash table and by using a good collision handling policy, but that makes the algorithm very complex to understand. The steps of the algorithm are as follows:

Let R be the first argument relation and S be the other.

1. Do step 1 of the algorithm given for the table representation in the dual application operation, on relation S on the SCHT1.

2. Do step 1 of the algorithm given for the table representation in the dual application operation, on relation R on the SCHT2.

3.   Start from the beginning of the linked list constructed in SCHT2 and proceed down in that linked list by skipping the records between the right individuals' records. For each right individual record found in this manner begin from the beginning of the linked list constructed in SCHT1 and proceed down in that linked list by skipping the left individuals' records between the right individuals' records.

a. For each pair of right individuals found above (one of the right individuals belongs to R and the other right individual belongs to S) call algorithm "pair" with those individuals as arguments. Create a new table record of the resulting relation, and put the pointer to the table of the relation (pair) constructed by algorithm "pair" into the "right" field of that record.

b. Find the SCHT record immediately following the right individual's record in SCHT2 (which belongs to the left individual that is in relation with that right individual under the relation R). Extract the individual being represented by that record, find the SCHT record immediately following the right individual's record in SCHT1 (which belongs to the left individual that is in relation with that right individual under the relation S), and extract the individual being represented by that record.

c. Call algorithm "pair" with the individuals found in step 3-b as arguments and establish the pointer to the

resulting relation constructed by algorithm "pair" in the "left" field of the new relation's record created in step 3-a.

4. Link the records of the new relation created above to each other, by their link fields as they are created.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As can be seen the algorithm is surprisingly simple and efficient relative to our previous algorithm. In fact the cost of that algorithm is much less than the cost of the previous algorithm. The worst case time complexity function of that algorithm can be written as:

$$f = K*p1 + L*p2 + M*n1*n2 + C$$

where:

p1 = The size of the relation S,

p2 = The size of the relation R,·

n1 = The cardinality of the RIM set of relation S,

n2 = The cardinality of the RIM set of relation R,

and, the first term corresponds to step 1, the second term corresponds to step 2, the third term corresponds to steps 3 and 4 of the algorithm. The constant C represents the number of memory references made by the housekeeping operations. Let the cardinalities of all the LEM and RIM sets of the argument relations be the same and equal to "n". We know that in the worst case:

$$p1 = p2 = n*n$$

Under these considerations, the above complexity function becomes:

    f = Z*n*n + C

where:

    Z = K + L + M

So we conclude that the cost of that algorithm is much less than the cost of our previous algorithm, and the algorithm has the worst case asymptotical time complexity behaviour of $O(n^2)$. Our previous algorithm is expensive because of the initialization of the incidence vector. Suppose the initialized memory block for the incidence vector is always available, and we do not have to initialize the incidence vector. Even under this condition our previous algorithm, by having a huge constant in front of the second degree term, is more expensive than this algorithm. We want to point out that this is the first operation for which the table representation allowed us to define a more efficient algorithm than the Hash-Incidence-Vector representation allowed. The weakness of the Hash-Incidence-Vector representation is that it requires the incidence vector to be initialized; on the other hand the incidence vector is structured enough to be pipelined especially for initialization. So for small relations with small incidence vectors, this deficiency is omitable.

First Ancestral (fan:R) :

This operation takes a relation identifier and produces the reflexive, transitive closure of that relation; the resulting relation is also called the first ancestral of the given relation.

Because the resulting relation will be reflexive, it will contain those tuples in which the left individual and the right individual are the same in addition to the tuples obtained by performing second ancestral operations on the original relation. This also implies that the LEM set and the RIM set of the resulting relation will be equal to the MEM set of the original relation. Note that in this case Warshall's algorithm can be applied without modification.

The algorithm for Hash-Incidence-Vector representation is as follows:

1. Find the record of the relation in the relation table by hashing with the relation identifier to the relation table.

2. Follow the PFLM field of that record and find the first left member's record.

3. By following the TASE links between the records, proceed down in the LEM set. For each record found in this manner, do the steps below:

a. Extract the individual represented by that record.

324

b. Hash to the LHT and the RHT with that individual under the new relation's identifier, "fan:R"; establish both the LEM record and the RIM record for that individual in the LHT and in the RHT respectively. If these are the first records created in this manner, mark them with pointer P and Q respectively. If after hashing to the LHT/RHT, a record of that individual is found to be present in the LHT/RHT, do not create LHT/RHT records for that individual. Link the records created in this manner by their TASE links in the LHT and in the RHT.

4. Continue with the RIM set of the original relation by beginning from the beginning of the RIM set (i.e., go to step 3).

5. In steps 3 and 4, keep a right members index count for the RIM set being constructed and increment it for each record created. Put the updated value of that count into the index field of the record created most recently. After the execution of steps 3 and 4, start from the beginning of the LEM set of the resulting relation and keep a LEM set index count. Beginning from 1, increment it by the last value of the RIM set index count for each record of the LEM set passed and put the updated value of this count into the "index" field of the current record.

6. Allocate a block of memory for the new incidence vector as large as:

$$\frac{(\text{RIM-set-index-count})^2}{C}$$

where C is the memory word length. Save the beginning address of this block.

7. Hash to the relation table under the new relation's identifier, "fan:R", where R is the identifier of the original relation. Establish the record of that relation in the relation table and put pointer P and Q into the PFLM and the PFRM fields of that record respectively. Put the last value of the RIM set index count into the !LEM| and the |RIM| fields and put the beginning address of the incidence vector into the BASE field of that record.

8. Start from the beginning of the LEM set of the original relation and proceed down in the LEM set record by record. For each record found do these steps:

a. Call the left individual being represented by the current LEM set record (found above) X. Reference the new relation with the tuple <X,X> and put 1 into the incidence vector entry found.

b. Start from the beginning of the RIM set of the original relation and find the records of the RIM set in turn by following the TASE links between the records.

c. For each tuple represented by the pair of records found in steps 8 and 8-b, reference the original relation's

326

incidence vector and reference the new relation. Find the corresponding incidence vector entry and copy the entry of the original incidence vector to the new incidence vector's corresponding entry.

9.   Execute step 6 of the second ancestral operation's algorithm on the new incidence vector. But in this case let the $|\text{RIM}|$ represent the cardinality of the RIM set of the new relation instead of the cardinality of the RIM set of the original relation (and similarly for the $|\text{LEM}|$).

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

In the worst case the relation may be a universal relation on its MEM set; that means the LEM and RIM sets of the relation are exactly the same as the MEM set of the relation. So the LEM, RIM and MEM sets of the relation have the common cardinality n. Under this consideration we write the worst case time complexity function of the algorithm as follows:

$$f = 2*K*n + M*(n^2) + N*(n^2)*(n/C) + D$$

where:

n = The common cardinality of the RIM and LEM sets of the original relation.

C = The memory word length.

K = The constant number of memory references made in step 3 (or 4) for each record copied.

327

M = The constant number of memory references made while copying each bit of the incidence vector.

N = The constant number of memory references made in each iteration of the outer-most "for" loop in step 6 of the second ancestral algorithm.

D = The constant number of memory references made by the housekeeping operations.

In the above function the first term corresponds to steps 3, 4 and 5, the second term corresponds to step 8, the third term corresponds to step 9, and the fifth term corresponds to the remaining steps of the algorithm.

The term which constitutes the asymptotical time complexity behaviour of the algorithm is the third term, which is the same as the term given in the complexity function of the second ancestral algorithm for the corresponding step (step 6).

Now, let $U = 2*K$ and $T = N/C$; then the complexity function becomes:

$$f = T*(n^3) + M*(n^2) + U*n + D$$

So by looking at the term with the largest exponent we conclude that the algorithm has the worst case asymptotical time complexity behaviour of $O(n^3)$. This algorithm is slightly more expensive than the algorithm for second ancestral operation, as it is expected, but both algorithms have the same asymptotical time complexity behaviour. We

328

want to point out that the constant of the second term is larger than the constant of the second term of the second ancestral operation's complexity function. The reason is, in this case we are also obtaining the reflexive closure of the relation. The same holds for the third terms.

As we know, while we are taking the reflexive closure of a relation, we have to add those tuples to the relation which have the property that the left component and right component are the same individuals. Because some tuples of this kind may already be present in the original relation, we have to prevent ourselves from duplication of tuples while taking the reflexive closure. As a result of this we can not copy the original incidence vector to the new incidence vector blindly, as was done in the second ancestral operation. So we had to define the algorithm in a different manner and this increased the constants in front of the second and third terms of the above complexity function.

The algorithm for the table representation is similar to the algorithm for the second ancestral operation, except we have to add those tuples in which the left component and the right component are the same, if those tuples are not already present in the relation resulting from the second ancestral operation. So the algorithm should detect if tuples of this kind are present in the transitive closure of the original relation. If they are not it should add those tuples to the

resulting relation. Thus the algorithm becomes more expensive than the algorithm for the second ancestral operation, so there is no need to define that extremely expensive algorithm here.

Final Members (final:R) :

This operation, given a relation, restricts the LEM set of the relation to the set:

$\{x|$ not $(x\varepsilon$ (init:R))$\}$ where R is the identifier of the given relation. So in the relational language we can write the equivalent expression as:

(-init:R)/R

So the result of this operation is a relation which has those left individuals that are not the initial members of the given relation as its LEM set members.

The algorithm for the Hash-Incidence-Vector representation is given below:

1. Get the relation's identifier.

2. Hash with that relation identifier to the relation table and find the record of the relation.

3. Follow the pointer found in the PFLM field of the relation's record and find the record of the first left member of the relation.

4. Hash to the RHT with that individual under the relation's identifier. If there exists a record for that individual in the RIM set, make a separate copy of the LEM

330

record (which is found in step 3) of that individual in the LHT, under the relation identifier "final:R", where R is the identifier of the original relation.

5. If there is no record present for that individual in the RIM set, do nothing.

6. Proceed in the LEM set of the relation by following the TASE links between the LEM (LHT) records. For each record found, repeat step 4 for the individual represented by that record until the LEM set of the relation is exhausted. While creating the new relation's records in the LHT link them to each other as they are created (i.e., construct LEM set of the new relation in the LHT). While the above steps are being performed keep a count, and increment it for each new LEM record created.

7. Make separate copies of the RIM set records of the original relation in the RHT under the new relation identifier by first following the PFRM pointer from the original relation's record in the relation table and then following the TASE links between the records. For each RHT record found hash into the RHT with the individual represented by this record under the new relation's identifier and establish a record in the RHT. Copy all the fields (except the TASE field) of the original RIM record to the new record's corresponding fields, and link the records created in the manner explained above, to each other as they

are created (i.e., reconstruct the RIM set of the original relation as the RIM set of the new relation).

8. Establish the new relation's record in the relation table under the new relation's identifier. Establish the pointers to the new LEM set and the new RIM set into the PFLM and PFRM fields of that record respectively. Copy the "base" field of the original relation's record into that record's "base" field. (Both relations share the same incidence vector). Copy the "|RIM|" field of the original relation's record into the "|RIM|" field of the new relation's record. Put the final value of the "count" into the "|LEM|" field of that record.

Note that, as we have done in the restriction operations, we make the new relation share the incidence vector of the original relation. Also, because we copied the records as they were (except the TASE fields), the contents of the index fields did not change, so the new relation can share the incidence vector with the original relation. The costly steps of that algorithm are steps 6 and 7. In step 6 we went through the LEM set of the original relation exhaustively and in step 7 we copied the RIM set of the original relation The worst case complexity function of that algorithm is given below:

$$f = K*m + L*n + C$$

where:

K = Constant number of memory references made for each
LEM element found in step 6.

L = Constant number of memory references made for each
RIM set element in step 7.

C = Constant number of memory references made in steps 1,
2, 3 and 8.

m = Cardinality of the LEM set (original relation).

n = Cardinality of the RIM set (original relation).

Let m = n and T = K + L, then the function becomes:

f = T*n + C

So we conclude that the algorithm has the worst case asymptotical time complexity behaviour of O(n).

Now we have to think how the same operation can be performed on the table representation. Because there exists the possibility of the repetition of the individuals in the left and right columns, it is obvious that the resulting algorithm will be more costly than our previous algorithm. We can reduce the cost of the algorithm by making use of the SCHT mechanism. The algorithm is given below:

1. Perform steps 1 and 2 of the algorithm given for the table representation in the "Initial members" operation. This time use SCHT instead of SHT.

2. Start from the beginning of the left column and for each individual found by following the link fields of the

records (and finding the left individual of each record pointed by the pointer in the "left" field of that record) hash into the SCHT. If no record is present in the SCHT for that individual, make a separate copy of the table record in which we found the individual in question. If it is the first table record created, mark it with a pointer.

3. Perform the disconnection operation in the SCHT as it was done in "initial members" operation to clean up the SCHT for the subsequent operations.

As we can see we go through the original relation three times exhaustively in steps 1 and 2. In the worst case the cardinality of the initial members set constructed in the SCHT may be equal to the cardinality of the LEM set, under this consideration the worst case time complexity function of this algorithm can be written as:

$f = K*p + M*p + N*p + L*n + C$

let $(K + M + N) = Z$; the function becomes:

$f = Z*p + L*n + C$

where $Z >= 3$. Now we have to explain the meanings of the constants and the variables shown in the first function in order to make the function clear.

The constants K and M are the same constants we defined in the "Initial members" operation case. The constant L represents the constant number of memory references for disconnecting each record from the SCHT. The constant N

334

represents the constant number of memory references made for each left individual found in step 2 of the algorithm. The constant C represents the constant number of memory references made by the housekeeping operations. The variables are defined as follows:

p = The relation size.

n = Cardinality of the LEM set.

As can be seen the constants tend to be large and we know that in the worst case (when the relation is equal to the cartesian product of its LEM and RIM set) p is equal to the product of the cardinalities of the LEM and RIM sets, so let's rewrite the complexity function under this consideration. Let n = m (where m is the cardinality of the RIM set). The function becomes:

f = Z * n*n + L*n + C

So we conclude that the algorithm has the worst case asymptotical time complexity behaviour of $O(n^2)$.

First Member (first:R) :

This operation, given a relation, finds the unique initial member of that relation, if there exists one and only one initial member of that relation. Otherwise the operation is undefined. The algorithm for the Hash-Incidence-Vector representation is given below:

335

1. Do steps 1 through 4 of the "initial members" operation's algorithm (for the Hash-Incidence-Vector representation).

2. Proceed down in the LEM set of the relation. For each individual of the LEM set hash to the RHT with the individual found. If there is no record for that individual in the RIM set of the relation then, record that individual and set a flag to true. Find the next individual's record in the LEM set of the relation, by following the TASE link field of the current record of LEM set. Continue to perform the same operation on the individual being represented by that record. If any subsequent operation results with another initial member (which can be detected by checking the flag) call the error routine, else continue to check until the LEM set of the relation is exhausted.

3. Return the individual recorded in step 2 if no error occurs.

In fact the above algorithm is the worst case algorithm that must be executed in the absence of the initial members set of the relation. So the first steps of the above algorithm should be:

1. Concatenate character string "init:" with the relation identifier.

2. Hash with the resulting identifier to the set table. If a record is present under that identifier, follow the PSS

336

field of that record, and find the first record of the set structure.

3.   Test if there exists any other record in this set; if so call the error routine, else follow the PML field of the set's record, find the individual and return that individual as the result of the operation.

4.   If no record is found in step 2 (in the set table) continue with the previous algorithm.

Since we are concerned with only the parts of the algorithms that cause the worst case behaviour, we can focus on the previous algorithm for the time complexity analysis.

Now we will do the worst case asymptotical time complexity analysis of this algorithm.

As can be seen, step 2 of the previous algorithm constitutes the asymptotical behaviour of that algorithm. The complexity function can be written as:

$f = K*n + C$

where constant K is the number of memory references made for each LEM set record found and constant C is the number of memory references made in step 1.

So we conclude that the worst case asymptotical time complexity behaviour of that algorithm is $O(n)$.

Now we have to define the algorithm for the table representation. We will again use the SCHT mechanism in order to make the algorithm efficient.   (We can use the same

337

argument we have used in the "initial members" operation about how inefficient the algorithm would be otherwise). The steps of the algorithm are given below:

1. Find the first record of the table.

2. Proceed in the table by following the link fields between the records. Extract the right individuals represented by the "right" field of each record, hash with each right individual into the SCHT, and establish its SCHT record in the SCHT. Link the SCHT records to each other by their TASE link fields as they are created.

3. After the right column is exhausted begin from the beginning of the relation again. Proceed down in the left column. For each left individual found in the same manner as it was done for the right individuals, hash into the SCHT to check if a record is already present for that individual in the SCHT (i.e., that is effectively looking up the right column in an efficient way because we established the individuals of the right column in the SCHT in step 2). The first time a record is found in the SCHT for a left individual record that individual and set a flag to true.

4. If in any of the subsequent repetitions of step 3, a record is found in the SCHT corresponding to a left individual (i.e., while flag is true) call the error routine. If the above situation does not occur until after the relation is exhausted (i.e., effectively until after the left column

is exhausted) return the recorded individual. If there does
not exist any initial members call the error routine.

5. Disconnect all the records from the SCHT (dispose
them).

Clearly steps 2 and 3 of this algorithm require one to go
through the relation once so the complexity function becomes:

f = 2*K*p + L*n + C

where Constant K is the number of memory references made for
each element of the right column and left column of the table
in step 2 and 3, Constant L is the number of memory
references made for each element of the RIM set, and constant
C is the number of memory references made by the housekeeping
operations. Variable "p" is the size of the relation, in
other words the number of tuples in the relation. Variable
"n" is the cardinality of the RIM set of the relation.

We know that in the worst case the relation may be equal
to the cartesian product of its LEM set and RIM set and,

p = n * n

where n is assumed to be the cardinality of both the LEM and
the RIM set of the relation. Let T = (2K + L), so under the
above consideration the worst case complexity function
becomes:

f = T*n*n + C

So we conclude that the algorithm has the worst case
asymptotical time complexity behaviour of $O(n^2)$.

THE COMPLEXITY FUNCTIONS WITH THE PREDICTED CONSTANTS

Function Application	HASH INCIDENCE VECTOR	TABLE
	15	$5n^2$
R!:x	$1.4n^2 + 36n + 28$	$12n^3 + 15$
Rc	$3n^2 + 36n + 35$	$4n^2 + 32$
unimg:R:x	$19n + 37$	$16n^2 + 14$
non:R	$0.02n^2 + 64n + 8$	$6n^3 + 50n^2 + 3n + 16$
R&S	$41n^2 + 51n + 42$	$7n^4 + 4n^2 + 1$
R-S	$87n^2 + 42$	$3n^4 + 4n^2 + 18$
R|S	$42n^2 + 20n + 52$	$14n^3 + 30n^2 + 18$
RVS (set)	$28n + 36$	
R/\S (set)	$28n + 36$	
R-S (set)	$28n + 36$	
init:R	$23n + 27$	$36n^2 + 16$
final:R	$46n + 30$	$50n^2 + 12n + 18$
first:R	$12n + 13$	$28n^2 + 12n + 11$
lem:R	$18n + 25$	$17n^2 + 12n + 14$
mem:R	$64n + 86$	$33n^2 + 15$
R\S	$64n + 43$	$36n^2 + 18$
C/R\C	$48n + 24$	$20n^2 + 18$
#:C	$2n + 14$	
RS	$0.9n^3 + 0.15n^2 + 38$	$14n^4 + 8n^2 + 18$

	HASH INCIDENCE VECTOR	TABLE
R‖S	$152n^4 + 0.15n^2 + 18$	$98n^2 + 18$
R#S	$0.15n^2 + 134n + 39$	$64n^2 + 18$
fan:R	$0.15n^3 + 42n^2 + 66n + 30$	
san:R	$0.15n^3 + 0.8n^2 + 34n + 30$	$30n^4 + 61n^2 + 18$

APPENDIX C

PREPROCESSING RULES

(1):	(R & S)c	=====>	Rc & Sc
(2):	(R | S)c	=====>	Rc | Sc
(3):	(R - S)c	=====>	Rc - Sc
(4):	(san:R)c	=====>	san:(Rc)
(5):	(fan:R)c	=====>	fan:(Rc)
(6):	(RS)c	=====>	(Sc)(Rc)
(7):	lem:(Rc)	=====>	rim: Rc
(8):	rim:(Rc)	=====>	lem: R
(9):	mem:(Rc)	=====>	mem: R
(10):	unimg:(Rc)	=====>	unimg':R
(11):	Rc\C	=====>	C/R
(12):	C/Rc	=====>	R\C
(13):	init:(Rc)	=====>	rim:R and not (lem:R)
(14):	init:(R)	=====>	lem:R and not (rim:R)
(15):	final:(Rc)	=====>	R \ (not(lem:R))
(16):	final:R	=====>	(lem:r and not (rim:R)) / R
(17):	(R & S)!:C	=====>	R!:C & S!:C
(18):	(R | S)!:C	=====>	R!:C | S!:C
(19):	(non:R)!:C	=====>	lem:R and not (R!:C)
(20):	lem:(R&S)	=====>	lem:R and lem:S
(21):	lem:(R|S)	=====>	lem:R or lem:S

```
(22):    rim:(R&S)         =====>    rim:R and rim:S

(23):    unimg(R&S)        =====>    unimg:R and unimg:S

(24):    unimg(R|S)        =====>    unimg:R or unimg:S

(25)     unimg(R-S)        =====>    unimg:R - unimg:S

(26):    lem:(fan:R)       =====>    mem:R

(27):    lem:(san:R)       =====>    lem:R

(28):    rim:(fan:R)       =====>    mem:R

(29):    rim:(san:R)       =====>    rim:R

(30):    unimg(R||S)       =====>    (R||S):

(31):    unimg(R#S)        =====>    (R#S) :

(32):    unimg'(R||S)      =====>    (R||S)c:

(33):    unimg'(R#S)       =====>    (R#S)c :

(34):    rim:(R\C)         =====>    rim:R and C

(35):    rim:(C/R)         =====>    rim:R

(36):    lem:(R\C)         =====>    lem:R

(37):    lem:(C/R)         =====>    lem:R and C

(38):    mem:R             =====>    lem:R or rim:R

(39):    lem:(final:R)     =====>    init:R

(40):    (non:R)c          =====>    non:(Rc)

(41):    rim:(final:R)     =====>    rim:R

(42):    mem:(final:R)     =====>    rim:R or (lem:R and not
                                     (rim:R))

(43):    init:(final:R)    =====>    init:R

(44):    R!:(rim:R)        =====>    lem :R

(45):    lem:(non:R)       =====>    (non:R)!:(rim:R)
```

```
(46):   rim:(non:R)      =====>   (non:Rc)!:(lem:R)

(47):   (unimg:(non:R)):x ==>    (lem:R - (unimg:R:x))

(48):   (unimg':(non:R)):x =>    (rim:R - (unimg':R:x))
```

APPENDIX D

MEMBERSHIP TEST ALGORITHMS CONTINUED

R!:C   (R is extensionally represented relation) :

Fr(unimg':R:z) -----set-----> Cl
Fr(C) ---Test-each-in----> Cl---any-->true{z is in the set.}

(non:R)!:C :

Fr(rim:R) ----------set---> Cl
Fr(unimg':R:z) -----set---> C2'
while C2'; Cl-C2' --set---> D
Fr(C) ---test-each-in-----> D ---any--> true{z is in the set}

(non:R)c!:C :

Fr(lem:R) ------------set---> Cl
Fr(unimg:R:z) --------set---> C2'
while C2'; Cl-C2' ----set---> D
Fr(C) ---test-each-in---> D ---any--> true{z is in the set}

```
unimg':(R||S):x :


Fr(unimg:R:left(z)) ----set---> Cl
Fr(unimg:S:right (z)) --set---> C2
left(x) --is--in--> Cl --tx---> true--tx-->varA
right(x)--is--in--> C2 --tx---> true--tx-->VarB
varA and varB ----tx--->true{z is in the set}


unimg':(R#S):x :


Fr(unimg':R:left(x))   ----set----> Cl
Fr(unimg':S:right(x)) ----set----> C2
Cl and C2 ---------------set----> D
z ---is--in-->D ---tx---> true{z is in the set}


unimg':(RS):x :


Fr(unimg':R:x) ---------set----> Cl
Fr(unimg :S:z) ---------set----> C2'
while C2'; Cl and C2' ---tx----> D'
while D'; isempty(D')----tx----> false{z is in the set}
```

```
lem:(R-S) :


Fr(unimg':R:z) --------set----> Cl
Fr(unimg':S:z) --------set----> C2'
while C2'; Cl - C2' ---tx----> D'
while D'; isempty(D') -tx----> false{z is in the set}


lem:(non:R) :


Fr(rim:R) ----------set---> Cl
Fr(unimg':R:z) -----set---> C2'
while C2'; Cl - C2'-set---> D'
while D'; isempty(D')tx---> false{z is in the set}


lem:(R||S) :


left(z) -----is--in---> lem:R --tx---> true --tx--->varA
right(z) ----is--in---> lem:S --tx---> true --tx--->varB
varA and varB ---tx---> true{z is in the set}


lem:(R#S) :

Fr(unimg':R:z) ---------set----> Cl
Fr(unimg':S:z) ---------set----> C2'
while C2'; Cl and C2' ---tx----> D'
while D': isempty(D') ---tx----> false{z is in the set}
```

347

```
lem:(RS) :


Fr(lem:S) ------------set----> Cl
Fr(unimg':S:z)-test-each-in-->Cl--any-->true{z is in the set}


rim:(R-S) :


Fr(unimg:R:z) ---------------set---> Cl
Fr(unimg:S:z) ---------------set---> C2'
while C2';  Cl - C2' --------set---> D'
while D'; isempty(D') --------tx---> false{z is in the set}


rim:(non:R) :


Fr(lem:R) -------------------set---> Cl
Fr(unimg:R:z) ---------------set---> C2'
while C2'; Cl - C2' ---------set---> D'
while D'; isempty(D') --------tx---> false{z is in the set}


rim:(R||S) :


left(z) ---is-in---> rim:R---tx--> true ---tx--> varA
right(z) --is-in---> rim:S---tx--> true ---tx--> varB
varA and varB -------tx----> true{z is in the set}
```

348

```
rim:(R#S) :


z ---is-in---> rim:R ---tx--> true ---tx--> varA
z ---is-in---> rim:S ---tx--> true ---tx--> varB
varA and varB ---tx---> true{z is in the set}


rim:(RS) :


Fr(rim:R) -------set----> Cl
Fr(unimg:S:z) --test-each-in--> Cl --any--> true{z is in the
                                set}
```

LIST OF REFERENCES

1. Henderson, P., <u>Functional Programming Application and Implementation</u>, Prentice-Hall, 1980, pp. 223-231.

2. MacLennan, B.J., <u>Introduction to Relational Programming</u>, Computer Science Department Technical Report NPS52-81-008, Naval Postgraduate School, 1981.

3. Backus, J., <u>Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs</u>, CACM 21, 8 (August 1978), pp. 613-641.

4. Baase, S., <u>Computer Algorithms Introduction to Design and Analysis</u>, Addison-Wesley Publishing Company, 1978, pp.219-224.

5. Aho, A. V., and Hopcroft, J. E., and Ullman, J. D., <u>The Design and Analys of Computer Algorithms</u>, Addison-Wesley Publishing Company, 1974, pp. 49-52.

BIBLIOGRAPHY

Baase, S., <u>Computer Algorithms Introduction to Design and Analysis</u>, Addison-Wesley Publishing Company, 1978.

Henderson, P., <u>Functional Programming Application and Implementation</u>, Prentice-Hall, 1980.

Horowitz, F., and Sahni, S., <u>Fundamentals of Data Structures</u>, Computer Science Press, Inc., 1976.

MacLennan, B. J., <u>Introduction to Relational Programming</u>, Computer Science Department Technical Report NPS52-81-008, Naval Postgraduate School, 1981.

Pratt, T.W., <u>Programming Languages Design and Implementation</u>, Prentice-Hall, 1975.

Stanat, D. F., and McAllister, D. F., <u>Discrete Mathematics In Computer Science</u>, Prentice-Hall, 1977.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, Virginia 22314

2. Library, Code 0142 2
   Naval Postgraduate School
   Monterey, California 93940

3. Department Chariman, Code 52 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93940

4. B. J. MacLennan, Code 52M1 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93940

5. D. R. Smith, Code 52Sc 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93940

6. Turk Deniz Kuvvetleri 5
   Egitim Daire Baskanligi
   Bakanliklar Ankara TURKEY

7. Ltjg Suha Futaci 2
   Ataturk Caddesi, Petek Apt
   Kat 3, Daire 9, Bursa TURKEY

8. Lt. A. Bresani 1
   Ministerio de Marina, Central
   de Procesamiento de Datos
   Lima, PERU