Courant Computer Science Report #15

September 1979

Data Structure Choice / Formal Differentiation Two Papers on Very High Level Program Optimization

Ssu-cheng Liu and Robert Paige

Courant Institute of Mathematical Sciences

Computer Science Department

New York University

Report No. NSO-15 prepared under Grant No. NSF-MCS-76-00116 from the National Science Foundation

COURANT COMPUTER SCIENCE PUBLICATIONS

COURANT COMPUTER SCIENCE NOTES

A101 ABRAHAMS, P. The PL/I Programming Language, 1979, 151 p.

C66 COCKE, J. & SCHWARTZ, J. Programming Languages and Their Compilers, 1970, 767 p.

D86 DAVIS, M. Computability, 1974, 248 p.

M72 MANACHER, G. ESPL: A Low-Level Language in the Style of Algol, 1971, 496 p.

M81 MULLISH, H. & GOLDSTEIN, M. A SETLB Primer, 1973, 201 p.

S91 SCHWARTZ, J. On Programming: An Interim Report on the SETL Project. Generalities; The SETL Language and Examples of Its Use. 1975, 675 p.

S99 SHAW, P. GYVE — A Programming Language for Protection and Control in a Concurrent Processing Environment, 1978, 668 p.

S100 SHAW, P. " Vol. 2, 1979, 600 p.

W78 WHITEHEAD, E.G., Jr. Combinatorial Algorithms, 1973, 104 p.

COURANT COMPUTER SCIENCE REPORTS

1 WARREN, H. Jr. ASL: A Proposed Variant of SETL, 1973, 326 p.

2 HOBBS, J. R. A Metalanguage for Expressing Grammatical Restrictions in Nodal Spans Parsing of Natural Language, 1974, 266 p.

3 TENENBAUM, A. Type Determination for Very High Level Languages, 1974, 171 p.

4 OWENS, P. A Comprehensive Survey of Parsing Algorithms for Programming Languages, ... 652+ p.

5 GEWIRTZ, W. Investigations in the Theory of Descriptive Complexity, 1974, 60 p.

6 MARKSTEIN, P. Operating System Specification Using Very High Level Dictions, 1975, 152 p.

7 GRISHMAN, R. (ed.) Directions in Artificial Intelligence: Natural Language Processing, 1975, 107 p.

8 GRISHMAN, R. A Survey of Syntactic Analysis Procedures for Natural Language, 1975, 94 p.

9 WEIMAN, CARL Scene Analysis: A Survey, 1975, 62 p.

10 RUBIN, N. A Hierarchical Technique for Mechanical Theorem Proving and Its Application to Programming Language Design, 1975, 172 p.

11 HOBBS, J.R. & ROSENSCHEIN, S.J. Making Computational Sense of Montague's Intensional Logic, 1977, 41 p.

12 DAVIS, M. & SCHWARTZ, J. Correct-Program Technology/Extensibility of Verifiers, with an Appendix by E. Deak, 1977, 146 p.

13 SEMENIUK, C. Groups with Solvable Word Problems, 1979, 77 p.

14 FABRI, J. Automatic Storage Optimization, 1979, 159 p. ..

15. LIU, S-C. & PAIGE, R. Data Structure Choice/Formal Differentiation. Two Papers on Very High Level Program Optimization, 1979, 658 p.

16 GOLDBERG, A. T. On the Complexity of the Satisfiability Problem, 1979, 85 p.

Notes: Available from Department LN. Prices on request.
Reports: Available from Ms. Lenora Greene. Nos. 1,3,4,6,7,8,10 available in xerox only..

COURANT INSTITUTE OF MATHEMATICAL SCIENCES
251 Mercer Street
New York, New York 10012

COURANT INSTITUTE OF MATHEMATICAL SCIENCES

Computer Science NSO-15

DATA STRUCTURE CHOICE / FORMAL DIFFERENTIATION

Ssu-cheng Liu and Robert Paige

September 1979

Report No. NSO-15 prepared under
Grant No. NSF-MCS-76-00116 from
the National Science Foundation

TABLE OF CONTENTS

Page

Automatic Data Structure Choice in SETL

by Ssu-cheng Liu 1

Expression Continuity and The Formal

Differentiation of Algorithms

by Robert Paige 269

AUTOMATIC DATA STRUCTURE CHOICE IN SETL

Ssu-cheng Liu

CONTENTS

		Page
PREFACE		2
CHAPTER		
1	INTRODUCTION	3
	1.1 Automatic Data Structure Choice System	7
	1.2 Related Work	10
	1.3 Review of Salient Features of the SETL Language	13
	1.4 Definitions	19
2	THE SETL BASING SYSTEM	22
	2.1 The Notion of "Basing"	22
	2.2 Data Structure for Based Representations	25
	2.3 Bases	33
	2.4 Details of the Basing Language	35
	2.5 A Case Study on the Application of Basings	41
3	AUTOMATIC DATA STRUCTURE CHOICE SYSTEM	53
	3.1 Essential Observation	53
	3.2 Fundamental Idea	55
	3.3 Overview of the System	58
	3.4 Phase I: Base Generation	61
	3.5 Phase II: Locate Emitting and Base Equivalencing	65
	3.6 Phase III: Locate Insertion	71
	3.7 Phase IV: Mode Determination	73
	3.8 Phase V: Refinement	75
	3.9 Supplementary Remarks	77
4	EXAMPLES	80
	4.1 Example 1: Tree Traversal	81
	4.2 Example 2: Spanning Tree	85
	4.3 Example 3: Huffman Coding	90
	4.4 Example 4: Maximum Flow	96
	4.5 Example 5: Interval Analysis	101
5	CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS	105
	5.1 Merging Rule	106
	5.2 Conversion of Representation Structure	108
	5.3 Conversion of Basings	108
	5.4 Conversion of Sparse Objects	109
	5.5 Multilevel Basings	110
	5.6 Colinked Bases	111
6	SETL CODE FOR THE DATA STRUCTURE CHOICE ALGORITHM	112
APPENDIX		
A	PRIMITIVE SETL OPERATIONS	249
B	ALPHABETICAL LISTING OF GLOBAL NAMES REFERENCED	252
BIBLIOGRAPHY		258

PREFACE

Abstract

SETL, like other very high level languages, emphasizes the use of abstract data structures such as sets and maps. Efficient data structuring of a SETL program is obtained by supplying to the language processor detailed descriptions of structural relations among program variables. These descriptions center around the concept of 'basings'. The use of basing declarations leads to efficient data layouts for the abstract structures of SETL. The concept of basings suggests techniques for choosing efficient data structures automatically by means of program analysis. The basing notion thus provides a unifying framework for automatic data structure choice. This paper describes a design for a demonstration system constructed within this framework. The manner in which such a system would apply to typical examples is illustrated. A SETL specification of the proposed automatic data structure choice scheme is also presented.

Acknowledgements

I am truly grateful for the help and support of the SETL project members and the Computer Science Department at New York University. I would like to take this opportunity to express my gratitude to all the following people who have helped in the development of this thesis: Professor Edmond Schonberg, who strongly influenced my ideas and provided valuable assistance, particularly in the early stage of my research; Micha Sharir, who checked my algorithms and made many useful comments; Arthur Grand, who constantly provided current information on the state of the SETL system; and last, and most important, Professor Jacob Schwartz, my advisor, without whose enthusiasm and patience this thesis would not have been finished.

My special thanks go to my wife, Ai-ju, for her encouragement and patience during my graduate years. And apologies are due to my son Oliver for my constant preoccupation with study.

CHAPTER 1 : INTRODUCTION

Research in automatic programming has over the years produced a number of increasingly powerful tools for the writers of software : symbolic assemblers, macro assemblers, text editors, debugging systems and algorithmic languages of greater and greater expressive power and conciseness. Ongoing efforts to design very high level languages, which incorporate powerful mathematical primitives and abstract information structure e.g., sets and relations, are one of the most important current aspects of this research. At the present time, a number of compiler/interpreter systems for languages of this level, such as PLANNER, MADCAP, VERS2, CONNIVER, LEAP and SETL, have already been developed. Results so far with such languages indicate that they do indeed greatly reduce programming effort, allow the programmer to tackle problems that would be intractable with lower level languages, and simplify the production of software.

However, until now, most efforts in this area have concentrated on language design and basic implementation ; little has yet been done to achieve efficient program execution. The inefficiency of existing implementations of languages of this level has been severe enough to restrict their use to a few research environments. This inefficiency mainly arises from two sources : generation

of unoptimized code and use of ill-chosen data structures.

By providing powerful semantic primitives and a comfortable syntax for combining these primitives, very high level languages can tempt the user into a style of programming which is highly inefficient if unoptimized. For example, in SETL, the task of determining the number of positive integers in a set S can be written :

```
N := #{ X∈S | X>0 } ;          (1)
```

Taken literally, this involves the construction of a set for the sole purpose of finding its cardinality. The loop :

```
N := 0 ; (∀ X∈S | X>0) N := N + 1 ;;    (2)
```

clearly achieves the same effect at a smaller cost. Yet a case can be made that (1), and the style of programming it embodies, represents a good use of the language. The intent of (1) is more apparent than that of (2); (1) is expressed more concisely, and can in fact be viewed as a specification for the 'lower-level' code fragment (2).

Similarly, SETL code to create the subset of positive numbers and the subset of negative numbers in a set S might be written as :

```
POS := { X∈S | X>0} ;   NEG := { X∈S | X<0} ;
```

The set S has to be scanned over twice if the source code

is interpreted or compiled directly without optimization ; which is certainly more inefficient than the code written in the lower level style :

```
POS := nl;  NEG := nl ;
(∀X∈S) if X > 0 then POS with X ;
          elseif X < 0 then NEG with X ; ;  end ∀X ;
```

These two examples show that the realization of a program written in a style fully utilizing a very high level language can never achieve reasonable efficiency without a powerful optimizer. The creation of redundant objects and the redundant looping through composite objects should in particular be avoided. Traditional code optimization techniques are in general useful for this purpose. Some of the prior work in this area is briefly reviewed in the section 2 of this chapter.

A second main source of inefficiency lies in the fact that, at the implementation level, the run-time support library, which realizes the high level semantic constructs of a very high level language, must use 'general' structures, which can support, with roughly even efficiency, all the various operations likely to be applied to the data objects of the various types provided by the language. Almost invariably, this general structure will not be the best choice for a specific application. SETL, for instance, realizes a set by a hash

table structure. Obviously, this is not the best structure for sets which are only subject to algebraic operations such as union and intersection. In this case, the use of bit string structures would be more appropriate. It is clear that to overcome this difficulty the language processor itself must have the capability of choosing efficient data structures to represent the abstract objects of the program and code sequences to realize the abstract operations to be performed on these objects. To accomplish this, we require a so-called automatic data structure choice system (subsequently denoted by 'ADSC system').

The research reported here is an attempt to demonstrate the feasibility of building such an ADSC system. We have designed a demonstration system, based on the notion of 'basing' (to be explained in the next chapter), to automatically choose data structures for variables of SETL programs. Since our system utilizes the information collected by other optimization techniques, it was designed as the final phase of a powerful SETL optimizer which incorporates a wide variety of useful, but better-established, optimization techniques. Though we have considered only a few abstract data structures which are available in SETL, we believe that the techniques used in the proposed ADSC system are generally applicable.

1.1 Automatic Data Structure Choice System

A reasonable approach to an ADSC system can be as follows :

(1) First, we have to choose a basic family of data structures. In attempting to regularize the process of data structure choice we do not ask whether all, or even many, of the data structures that might be used by an experienced programmer can be duplicated automatically. Rather, we try to find some narrow subfamily of the family of all possible data structure choices, doing this in a way which guarantees that some choice in our subfamily is an adequate replacement for any choice which a programmer is likely to make. There is no doubt that a poor initial choice of subfamily may have a severely deleterious effect on the execution efficiency of the operations defined in the language. Choosing a proper family of data structures is therefore quite important. In making this choice, we must consider at least the operations to be applied to each of our abstract structures, the relative frequencies of these operations, and the relative importance of conserving time versus space.

(2) Having chosen a basic family of data structures, we can estimate the execution speed of each possible operation on these different data structures. This

provides fundamental information about each individual structure that is considered in the process of making an eventual data structure choice. A difficulty in this process arises from the fact that the execution speed of an operation may be a function both of the data structures of its operands and of the expected size of each operand. These sizes are not deducible directly from the program text. Inhomogeneity of the components of a composite object also makes speed problems unsolvable in some cases. Hence some kind of approximation has to be applied at this step. However, this approximation should not blur our analysis so much that we are not able to distinguish between different data structures.

(3) Next, we have to construct a fact collector to analyze source programs. The fact collector must be able to derive the information which is relevant to the structure choice algorithm that we intend to use. Standard global program analysis techniques such as interval analysis, data flow analysis, value flow analysis and plausible relation deduction techniques, etc., are all valuable here.

(4) As a final step, we must design a structure choice algorithm. This is the heart of an ADSC system. Here, the speed functions or the characteristics of data structures developed in the previous step and gathered

'facts' concerning each individual program must be used in order to minimize the total cost of running the program. Of course, the design of such a decision-making algorithm will be somwhat ad hoc due to the fact that the facts which enter into good data structure choice are not all accessible to a program analysis routine. We may have to make some crudely empirical decisions, seeking guidance in this by manually translating a representative variety of algorithms written in source language into equivalent but efficient codes in the target lower level language. If such translation is done as systematically as possible and in a highly 'self-conscious' way, then by noting the facts which repeatedly appear relevant in the process reasonably good mechanical data structure choice algorithms may be achieved.

During the design of our ADSC system, we have used this approach. A first step in our approach was to introduce a 'basing system', which in effect defines the subfamily of data structures with which we work. Initially, a declaration language which allows mannual definitions of basings was provided ; see Schwartz[1971a,1971b] and Schwartz[1976a,1976b]. This gave an extremely useful tool to experiment with data structure choice and to explore various styles of choice. Then, exploiting this experience, we have designed our automatic basing choice system. Chapter 2 of this thesis

will present the 'basing system' on which our work rests, while chapter 3 will describe the automatic data choice algorithm.

The SETL optimizer has been assumed to be available as the program analysis component of our system. This gives us the global information required by our ADSC system. In this thesis, no detailed discussion about the analysis techniques used in this optimizer will be given, see however Vanek[1976b], Sharir[1977] and Grand[1978].

1.2 Related Work

In this section we will review prior work related to our research.

Tenenbaum[1974] shows how to analyze SETL programs automatically to determine the types of the variables used in them. For this purpose, he builds a data type lattice with 'conjunction' and 'disjunction' operations. A revised version of this data type finder is described by Vanek[1976a]. The automatic data structure choice system presented in this paper will utilize the information provided by such an automatic 'type finder'.

The two part paper by Schwartz[1975c] introduces a number of analysis and optimization techniques for general

very high level languages, in particular for SETL. One of these techniques is value flow analysis, which generalizes conventional data flow analysis to structured data objects. A second technique introduced in this paper is copy optimization, which allows us to derive criteria under which we can modify existing data objects in place rather than generate new copies when they are logically modified. A related copy optimization technique first described by Dewar[1977] has been implemented in the current SETL optimizer. Finally, a technique of inclusion and membership determination which can provide valuable information to data structure choice was described.

Low[1974] describes an interesting approach to selection of representations for particular data types within the framwork of a small set of pre-selected data structure alternatives. The choice is made via a cost analysis of each alternative for the operations actually performed in the program. The cost of a program is calculated in terms of execution time and required space. Combinatorial explosion in the calculation of program cost is avoided by insisting that all variables (of the same type) subject to a single operation have the same representation structure. Low[1978] enhances this idea and gives an example and overview. A shortcoming in this approach is that logical relationships among program variables, which should play an important role in the

process to choose proper data structures for program variables, are completely ignored.

Bruce[1976] describes an APL optimizer incorporating various established optimization techniques. The program performance benefit of various possible transformations is estimated. Determination of a realization of the target program which gains maximum benefit is attempted. Combinatorial explosion difficulties in this process are avoided by a look-ahead scheme. The work of this optimizer is structured to avoid the creation of uncessary temporaries, in particular temporaries with large aggregate values. Loop optimization techniques such as loop paralleling, loop switching and loop jamming are employed. A simplified copy optimization principle which allows arrays to be used destructively is also included. No attempt to select optimal data structures is made.

Rovner[1976] has extended Low's work to finding implementations for associative data structures and accesses. Redundant representations for data structures are allowed. Rovner uses Low's hill climbing approach. Some additional heuristics about cost trade-off are added to the applicable conditions.

Kant[1977] describes a system LIBRA which aims to identify an efficient set of implementations for abstract

constructs in a very high level program. The high level constructs are realized by step-by-step refinement of partially refined programs (called program descriptions). Within this framwork, three basic issuses are addressed : picking a program description to refine, picking a particular statement or group of statements to refine and selecting a refinement for the chosen statement. At each level of selection, a set of heuristic rules is applied first, then follows a cost analysis. If the outcome of cost analysis is not clear, seperate program descriptions are set up to test several possibilities. The refinement process continues until a program in the target language is generated. In this system, multiple representations and the use of different representations for a variable in different parts of a program are allowed.

1.3 Review of Salient Features of the SETL Language

Before starting to describe our system, we shall review some of the important features of the language, SETL, that we are going to deal with. A general survey of this language is given by Kennedy and Schwartz[1975]. For a detailed account, see Schwartz[1973]. Several semantic and syntatic changes made recently in the new version of the language are summarized in Schonberg[1976]. For complete language description and programming reference, see Dewar[1975]. Appendix A lists most of the primitive

operations of SETL, and shows the syntactic form in which they are written.

SETL is a very high level, general purpose language based upon the dictions and semantic notions of the theory of sets. Both atomic and composite data types are supported. Atomics include the data types commonly found in most programming languages, such as integers, reals, bit strings of which boolean values are a special case, character strings, subroutines and functions.

Sets and tuples are two basic composite data types. Sets are the important objects in the language, whose uses characterize the semantics of the language. A set is an unordered finite collection of distinct SETL objects, and thus may contain atoms, tuples and other sets. Sets may be formed by enumeration, e.g. {1,2,3}, or by using a general set-former construction. The general set-former has the form

```
{ EXP : RANGE | COND }
```

where EXP is a genral expression, RANGE describes the iterative operation which calculates successive values of EXP, and COND specifies which of these values shall actually become members of the set being built.

A tuple is an ordered sequence of SETL objects. Two

tuples are equal if all their components are equal. Tuples, like sets, are built by explicit enumeration or by means of tuple-former expression,

```
[ EXP : RANGE | COND ]
```

which is analogous to a set-former expression. Sets and tuples can be nested freely as components and members of each other to any depth and can be entirely inhomogeneous.

Maps, which are special types of sets, play a special role. Maps are functions in the sense of set theory, i.e. sets of ordered pairs [X,Y] (i.e. tuples of length of 2), where X is an element of the domain of F, and Y is the corresponding element in the range. SETL allows such sets of pairs to be used as 'tabular functions' or relations. Maps can be manipulated in terms of their set structure, i.e. as sets of pairs. However, and most importantly, functional retrieval and storage operations can be applied to them. If F is such a map, then F(X) yields the second element of the unique pair in F, whose first element is X.

Maps need not be single-valued. A map may contain several pairs which have the same first element, in which case the map is called a multi-valued map. If F contains both [X,Y] and [X,Z], then the expression F(X) is undefined. However, the set of all values into which a map sends a given element X of its domain can be retrieved

by using the expression F{X}. In this case, F{X} yields the set {Y,Z}. If F is single-valued at X, F{X} yields a singleton set.

An additional range retrieval operation is provided for maps. If F is a map and S is a set then the expression F[S] is the set of images of elements of S under F, i.e. denotes the set

{ Y : X∈S,Y∈F{X} }

Because of the essential role they play in SETL, sets and maps are the central objects studied in this paper.

The undefined atom OM is a particular constant related to various SETL operations. OM is not allowed to be a member of any set but can be a component of a tuple. It is invalid in most contexts within expressions but can appear in an equality test. It is the valid result of several operations on sets and tuples. In particular, (a) if F is a map then the expression F(X) yields OM if F has not been defined or is multiply defined on X, (b) it is the value of an iterator variable at the end of an iteration, and (c) it is obtained when extracting an arbitrary element from the empty set.

The control structure of SETL is largely conventional and tends to follow ALGOL 60. Conditional statements and

expressions are provided, as are if-then-else clause, while-loops, cases, subprocedures and functions. 'quit' and 'continue' statements are included for abnormal loop control ; 'quit' causes control to leave the loop and 'continue' returns control to the beginning of the loop. A quit or continue statement is applied to the innermost loop with opening tokens matching the tokens following the keyword 'quit' or 'continue'.

The least familar SETL control form is the iterator-over-set, which is written as

(∀X1∈E1,...,Xn∈En | C(X1,...,Xn)) block ; end ∀ ;

Where E1,...,En are expressions with set values, C is a boolean expression of X1,...,Xn, and 'block' is any sequence of statements. This iterator expression executes 'block' repeatedly, once for each group X1,...,Xn of variable values belonging to E1,...,En respectively and satisfying the boolean condition C(X1,...,Xn). Convenient syntax to iterate over a map is also provided : in the iterator '∀Y:=F(X)', X varies over the domain of the map F, and Y receives the corresponding range element.

Functions and subroutines can be recursive. Parameters are passed by value without return, i.e., the values of the actual arguments passed to a function or subroutine are not changed in the calling routine. No

value can be returned from a subroutine. The value to be returned from a function must be specified in a return instruction.

Format is free, and statements are punctuated by semicolons. The PL/1 comment convention is employed. The abbreviated statement 'X op Y ;' stands for 'X := X op Y ;' A special primitive 'from', such that statement 'X from Y ;' is synonymous with 'Y := arb S ; S less Y ;', is also provided. A front-end macroprocessor is included as a convience in the SETL compiler. Macro definitions have the form

```
macro NAME (NAMLIST1 ; NAMLIST2 )  text  end NAME ;
```

Where NAMELIST1 contains the macro's arguments, and NAMELIST2 is a list of names for which new identifiers are generated for corresponding names in the macro body. Both namelists are optional.

A SETL program consists of a set of seperately compiled 'modules', each of which contains a set of functions and subprocedures. Variables are local by default but may be declared global to a module. Global variables may be made 'public', allowing them to be included in other modules. A user can determine that certain variables are stored statically while others are stacked on entry to a routine.

Every program must contian a module called MAIN. This module should contain a block of instructions which forms the main program. A simple form of initialisation block is also implemented. Only static variables can be initialized ; this intialization is performed before the start of execution.

1.4 Definitions

A number of terms which facilitate later discussion are defined in this section.

Occurrences :

An occurrence is a use or definition of a program variable.

Ovariables :

An ovariable is an occurrence at which the variable is assigned a new value.

Ivariables :

An ivariable is an occurrence at which the value of a variable is retrieved. For example, in the instruction 'X:=X+1;', the X in the left hand side is an ovariable while the X in the right hand side is an ivariable.

FFROM map :

FFROM is one of our main data flow maps. FFROM(OI) maps an occurrence OI of a variable V to the set of occurrences of V which can be reached from OI throurh a V-clear path, where a V-clear path means a path on which there are no occurrences of V. Note that the transitive closure of FFROM gives the traditional 'definition-use chain'.

BFROM map :

The map BFROM is the inverse of FFROM.

CRTHIS map :

This is a value flow mapping. CRTHIS(OI) maps a variable occurrence OI to its creation points, i.e., the set of all ovariables whose evaluation can create an object which at some moment in the execution of the program becomes the current value of OI. For example,

```
L1 : Y := X + 1 ;
L2 : S with Y ;
L3 : Z from S ;
L4 : U := Z ;
```

If we use the notation Vi to denote the occurrence of the variable V at the instruction I, then some of the relations among the variable occurrences in these three instructions will be

FFROM(Y1) = {Y2}, FFROM(S2) = {S3},

BFROM{Y2} = {Y1}, BFROM{S3} = {S2},

Y1 ∈ CRTHIS{U4}

PS-CRTHIS map :

This is the value flow mapping to be used in our data structure choice algorithm. PS-CRTHIS{OI} maps a variable occurrence OI to the pseudo creation points of OI, i.e., the set of variable occurrences which are the ovariables of value creation or value retrieval instructions and whose values can be transmitted to OI through simple assignment instructions. This map is similar to CRTHIS map but it does not link the occurrences whose values may be transmitted from one to the other through a series of value insertion into and value extraction from composite objects. For example, in the above example, the pseudo creation point of the U appearing at L4 will be the Z appearing at L3, while its creation point is the Y appearing at L1.

CHAPTER 2 : THE SETL BASING SYSTEM

In our approach to an ADSC system, we have used the semi-automatic data structuring system provided in the presently implemented SETL language as a stepping stone. In this SETL 'basing language', efficient data structures for the objects of a SETL program are defined by supplying the language processor with detailed declarations of structural relations among these objects. We regard the definition and implementation of this 'basing language' as two essential preliminary steps to our ADSC system. These two steps determine the basic family of data structures which we must consider, and give us a systematic language for characterizing data structures. Two succeeding steps - designing a data structure choice algorithm and the fact collector on which it rests then become the major issues to be studied in designing our total system.

In this chapter, we will describe the basing system, clarify the fundamental notions which it embedies, and illustrate its application.

2.1 The Notion of 'Basing'

In the absence of user-supplied declarations, the SETL processor chooses, for the sets and maps appearing in a

program, a default representation which is reasonably efficient for most of the primitive set operations commonly invoked. However, it is clear that very large gains in efficiency can be obtained if program variables are represented in a way which depends on the specific operations into which they enter. To give an obvious example, if sets S1 and S2 are known to be subsets of some other set B, then bit-vector representations for S1 and S2 (where an on-bit position indicates the presence of a given element of B in the corresponding subset) can be very advantageous if the intersection operation, S1*S2, is to be performed frequently.

Generalising the fundamental technique apparent here we say that an object X is based on another one B, if X is represented in some special, abbreviated form such that the presence of B is required for the full description of X. In this case B is said to be a 'base'.

In the present SETL system, two kinds of basings have been introduced.

(1) Member Basings - This scheme enhances the efficiency of SETL (which is a value language), by using pointer mechanisms internally. Whenever the value V of a variable X is known, either by declaration or by some decision made by the optimizer, to be an element of another set B (called the base), a pointer to the element of the same

value in B, instead of the value V itself, is kept in X. This technique allows us to save substantial execution time whenever internal 'locate' operations are required in the subsequent uses of the variable X. With such a pointer available, an indirect reference can replace a more expensive series of searches and equality tests. This can be extremely significant, especially when the value of X is a composite object.

(2) Domain Basings - When a set S is known to be a subset of another set B (which again we call the base of S), it can be represented by a collection of bits associated with B, each indicating whether a particular element of B is in S. These bits can be stored either locally with the elements of B, or remotely, i.e., the whole collection of bits can be stored as a bit string and each element of B can be supplied with an index which can be used to address all such 'remote bit strings'. A common advantage of this kind of structure is that both the 'local bit' and the 'remote bit' representations can save substantial space. Moreover, remote representation can speed up boolean operations on sets very greatly. The same approach can also be extended to maps whose domains are known to be subsets of the base. In this case, the collection of bits is replaced by a collection of map value pointers.

In summary, 'basing' which introduces indexing and pointer notions at the impelementation level of the SETL

system will play a central role in our approach.

2.2 Data Structure for Based Representations

In order to make clear the efficiency gains obtainable in the presence of basings, the concrete representations used for based objects are discussed in this section.

In the absence of declarations, the fundamental structure used to represent a set in SETL is a breathing hash-table, i.e., a hash-table whose table size is adjusted dynamically in order to keep the length of clash-lists approximately constant, so as to guarantee that the membership operation is always performed in a time which is independent of the size of the set being searched. Since in the standard SETL situation, map retrieval, set union and intersection all involve internal membership tests, a significant part of the execution cost of an undeclared SETL program is roughly proportional to the total number of hash-search operations performed (here we disregard the overhead involved in reallocating hash-tables). The efficiency advantage secured by the use of based representation is therefore seen to result from the possibility of replacing these hash-search operations by simpler code sequences, typically involving only one or two indexing operations, and from the possibility of using bit-parallel operations in some favorable cases.

The possible candidates for an extended library of set representations are numerous. The structures selected for incorporation into our library reflect this potential variety, but are compromised by the need to keep our library down to a manageable size, and our subjective judgement concerning the most important language constructs to optimize. The most significant of the structures we use are as follows.

If a map F is known to be based on B, and X is known to be an element of B, then the value of F(X) may be stored as part of the element block in B which represents X. To do this, we allocate, in each base element block, fields which will hold the values of some of the maps which are defined on these elements. Successive fields in this block correspond to various maps F1,F2....Fn whose domains are known to be subsets of B. When this representation is used, retrieval or assignment of F(X) becomes an indexing operation which uses an offset associated with F (known at compile time) to access an appropriate field in the element block for X. The representation used for X must then contain a pointer to the element block in B which represents X. Note, however, that in dealing with elements not known to belong to B, we must be able to locate them in B by using a standard hash-search procedure. Therefore in most cases B must have most of the hash-table structure of standard unbased

sets.

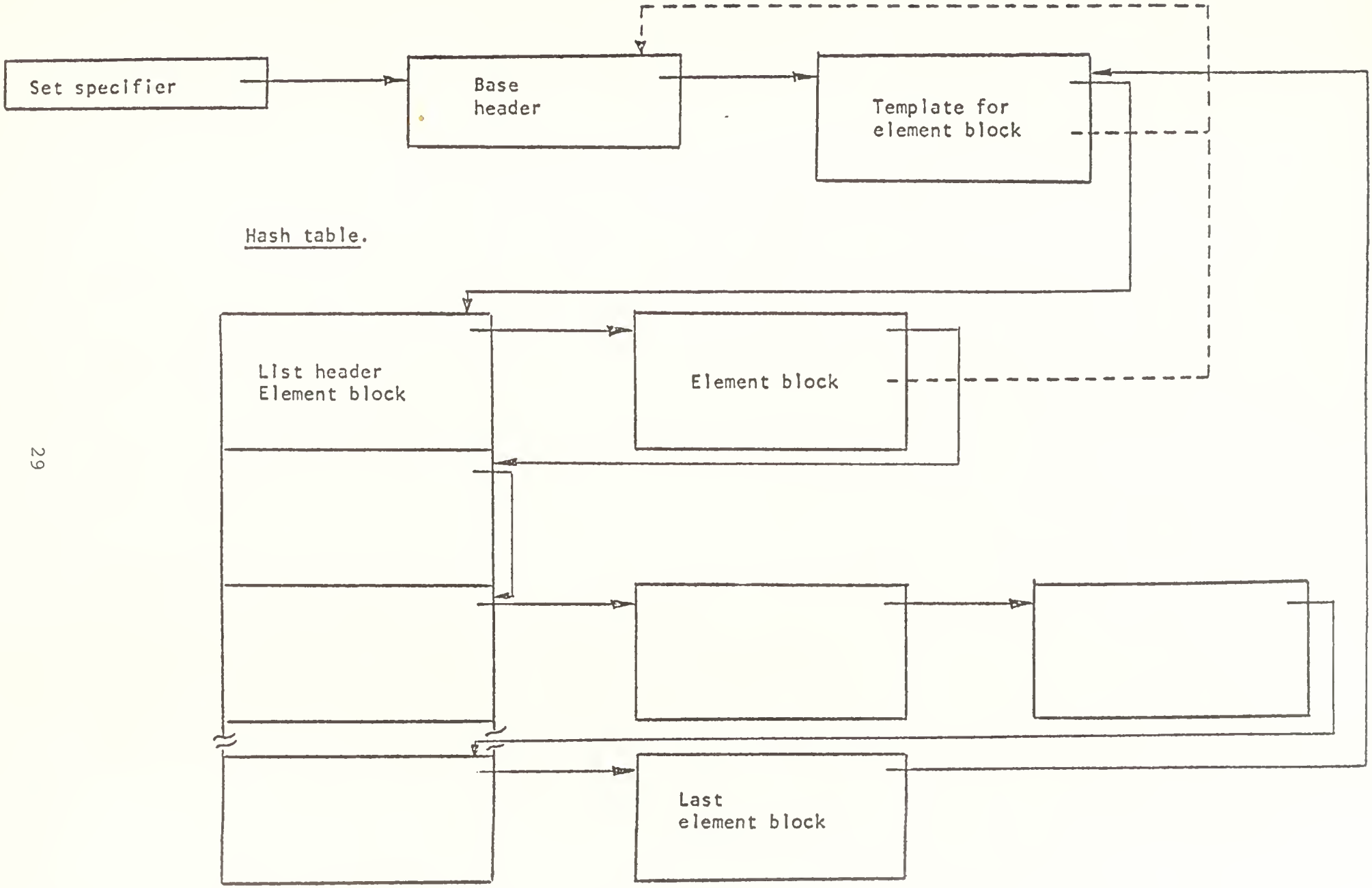
The map representation described above can be said to be local : map values are directly attached to the representation of elements of the map domain. This representation optimizes retrieval operations, but is awkward for global operations such as iteraton and copying, because of the distributed nature of the representation. An alternative representation, which is equivalent in amount of storage used and only slightly less efficient for retrievals, is available if we store the range of a map as a tuple, and incorporate a single index integer in each element of the map domain (i.e., of the base), using this index to access the tuple. The index is incremented sequentially whenever a new value is made an element of the base. Suppose the object X, $X \in S$, has index I. Then the value of F(X) is found by retrieving TF(I), where TF is the tuple representing F. This representation, which is a kind of dual to the local representation described previously, is said to be 'remote', and F is said to be a 'remote map' based on B.

Similar local and remote representations exist for subsets. If a set S is addressed only by insertions, deletions, and cardinality checks, then it can advantageously be represented by individual bits attached to the elements of a base B. If X is an element of B,

then the test 'X in S' is performed by examining that single bit in the element block of X which determines membership in S. Since we may want several subsets to be represented in this fashion, each element block in a base is allowed to contain a field whose i-th bit indicates membership of the corresponding element of B in the i-th subset based on B. Subsets represented in this way are said to be local subsets.

Local subsets support efficient insertion, deletion, membership test and cardinality check operations, but global operations such as union and intersection are inefficient when applied to local subsets. For such operations, bit-vectors, which can make use of hardware bit-parallel operations, are a more appropriate choice. For sets with this representation, the index i attached to element X of the base B (described above in connection with remote maps) is also used to index the representing bit-vector ; if bit i is on in the bit-vector representation of S, it indicates that $X \in S$. Membership tests addressing such structures are slightly slower than membership tests for local subsets, as one additional indexing operation is involved.

The data structures described so far are in general more compact than their unbased counterparts. There are cases, however, in which they may lead to very inefficient



g. 1. General Structure of base sets. Each clash list chains back to the hash table, to speed up iteration. The base header and template block are disjoint from the hash-table proper, to simplify table reallocation ("breathing").

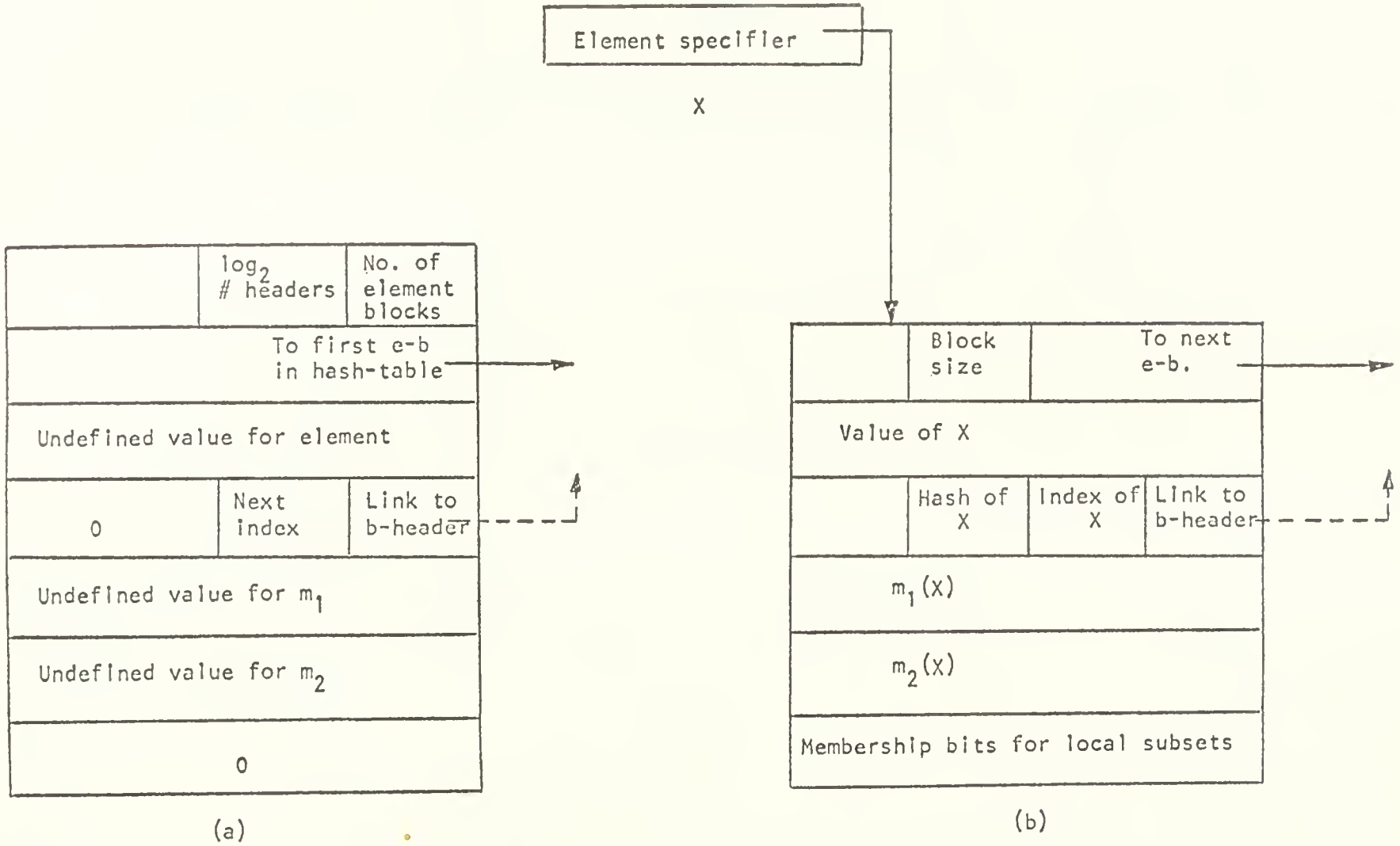


Fig. 2. a) Structure of template block for a base set B. Two local maps are defined on B.
b) Structure of element block in B, for X:εB. Only the most important fields are labelled.

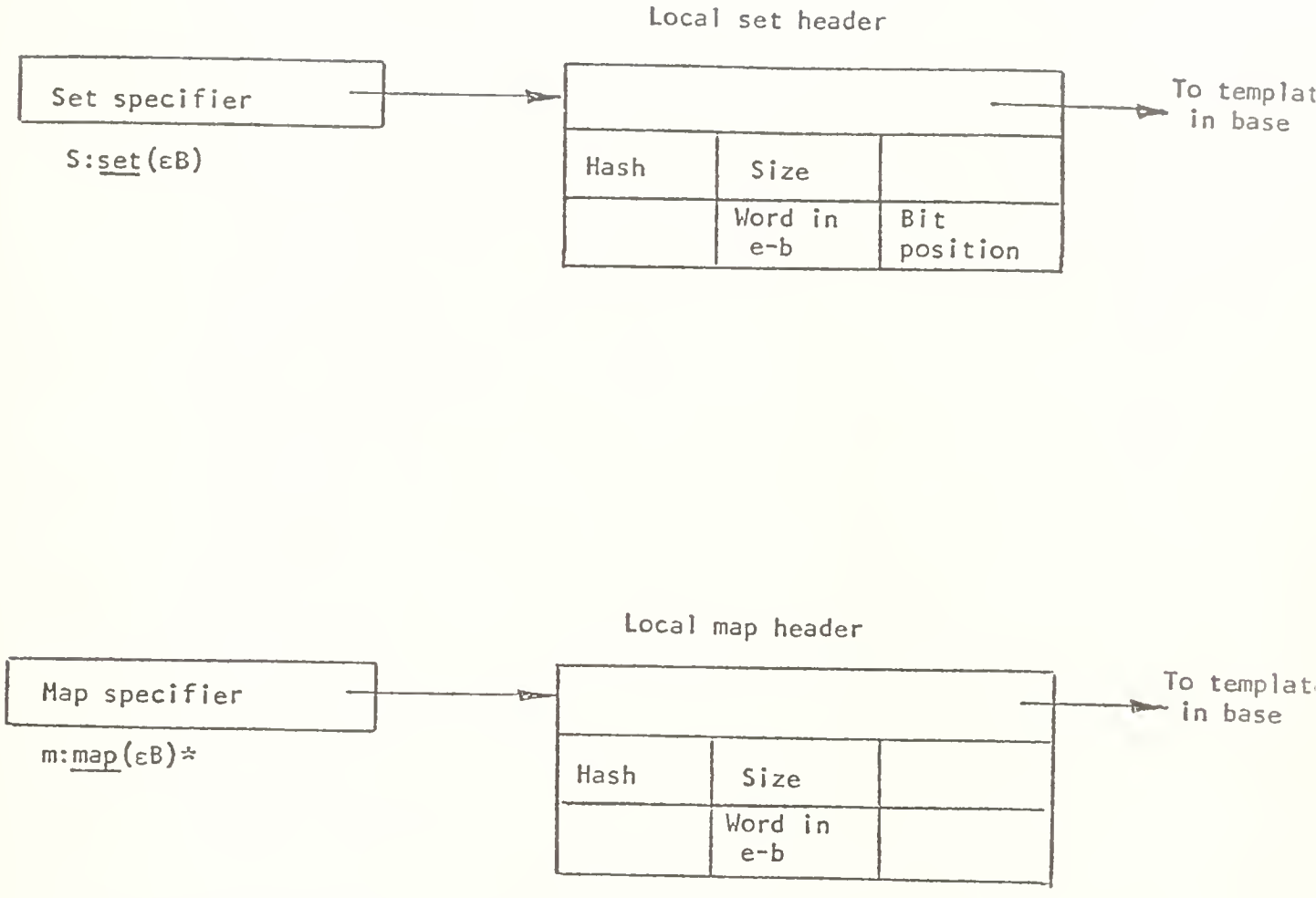


Fig. 3. Representation for local based objects.

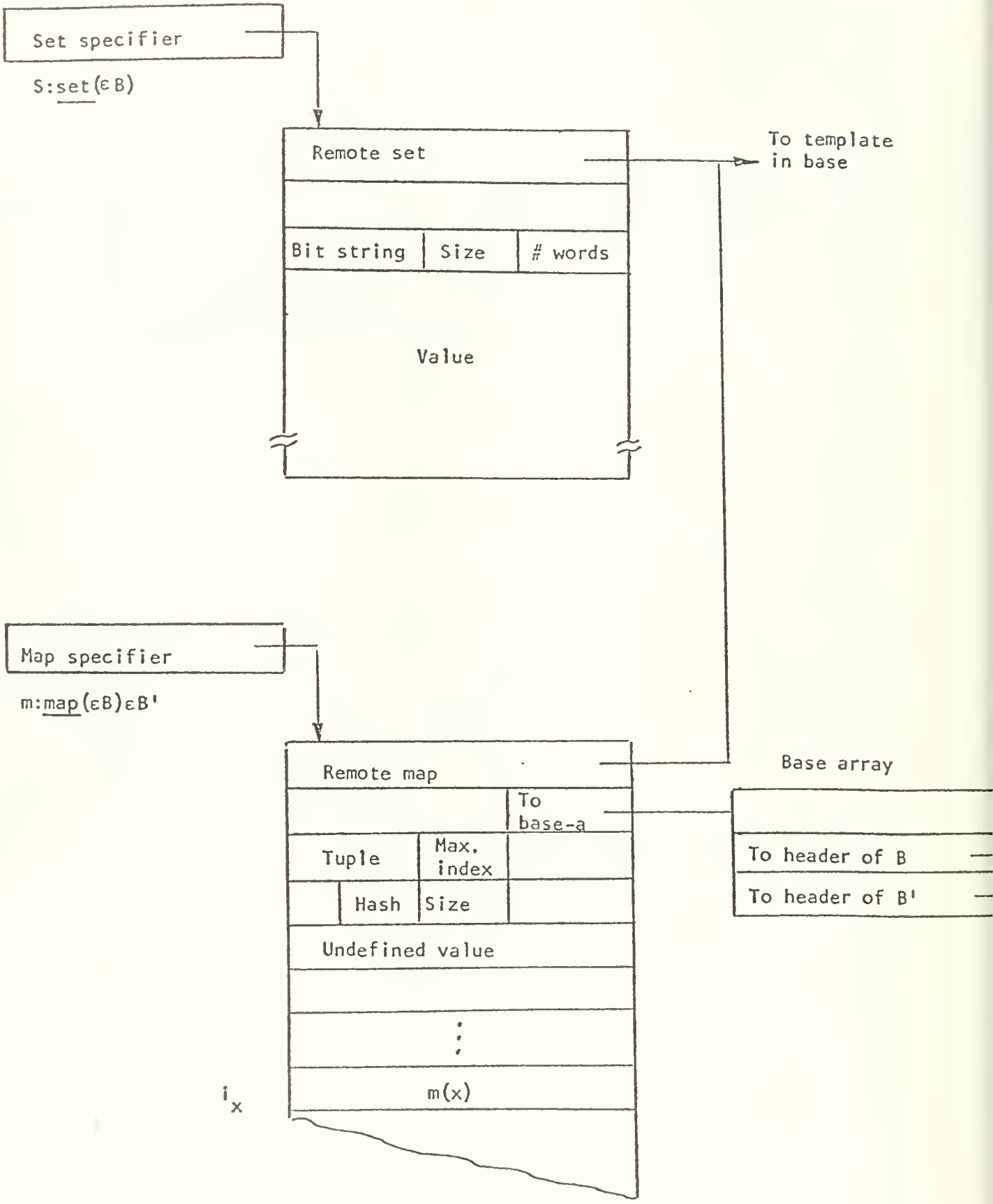


Fig. 4. Representations for remote based objects.
The base array contains links to the bases of based objects. It is used during execution-time mode conversions.

storage uses. If a based map (on B) is defined over a small subset SB of B , then #B-#SB words (one of which is allocated for each element block in B) will be wasted. In this case, it can be preferable to represent F as a standard hash-table, retaining however the '∈B' mode for the elements in its domain. This leads to so-called sparse representation. The sparse representations are still slightly more efficient than completely unbased ones, in terms of accessing : if X is ∈B and F is a sparse map domain based on B, then the hash-code for X is stored in its element-block in B, and need not be recomputed to access the hash-table for F when retrieving F(X) (this is because the hash code of a value is defined in a system-wide invariant way). The same technique can also be applied to subsets. Sets represented in this fashion are called sparse sets. The specific way a based object is ultimately represented (locally, remotely or sparsely) will be called the 'representation attribute' of the based object.

The considerations that we have just described lead to the structures pictured in figures 1 to 4. Further details can be found in Dewar et al[1977b].

2.3 Bases

The data structures described in the previous section

derive their usefulness from the pointer and index mechanisms which they implicitly make available. However, use of these efficiency- oriented pointer mechanisms creates the potential for conflict with the strict value semantics of SETL. For example, if X has been declared '$\in$B', and F(X) has been given a value once, subsequent deletion of X from B might be feared to have unanticipated side- effects : should we consider F(X) to still be defined ? If a set S is based on B and an element X of B was in S, should we say that the domain basing of S on B is invalid after X has been deleted from B ? To permit such side-effects is unacceptable since it would imply different semantics for progams, depending on whether based representations had or had not been declared. To avoid this, we insist that bases are not program variables, i.e., they are not explicitly created or modified by the user's code. The value of a base is defined only by the collection of all objects based on it. In a 'declared' program (i.e., one in which based representaions have been declared for variables) bases are built and updated in response to operations which create objects declared to be based on them. For example, if the declaration X$\in$B has been supplied, then whenever X receives a value which can not be determined a priori to be already in B, a hash-search in B is performed, and if the value of X is not already there, it is inserted into B. This implies that in the absence of some compaction

mechanism, bases can only grow monotonically during program execution. It also means that much of the cost of using based representations is incurred in the build-up of bases.

Note that when a user introduces basing declarations in his program, he will define bases using names which are distinct from any identifiers already present in his program ; but these bases may actually turn out to be identical in value to some actual program variables. For example, the variable S may be declared to be domain based on B, but it might be possible to ascertain that S=B, because B receives elements only when elements are added to S and no element is ever deleted from S. A SETL compiler able to recognize this might choose to treat S itself as a base. This illustrates a very general principle of our basing system : the processor will use pointer mechanisms as far as it safely can but the SETL value semantics visible at the user level will be preserved faithfully.

2.4 Details of the Basing Language

We shall now describe the syntax and semantics of basing declarations in more detail.

The generic term 'mode' is used in our system to refer

to the total information defining the data representation of a program variable. For each of the modes which we admit, a symbolic notation is introduced. The family of notations which thereby arise constitutes our data structure representation language, or basing language. Among these modes, the modes related to bases play the most important roles, as explained above.

Each variable in a SETL program can be declared to have a mode. A mode descriptor specifies the SETL type of a variable, and in addition, gives complete or partial structural information about it, e.g., its size and its relationship to other variables. Once a variable is declared, its mode stays static, i.e., can never be changed. The modes of undeclared variables are determined automatically by the language processor, but may be left 'general'. A related set of rules also determine the mode of compiler-generated temporaries.

In general, mode descriptors can be classified into four categories : 1) primitive modes, 2) bases, 3) derived modes and 4) composite modes.

2.4.1 Primitive Modes

Primitive modes correspond to the primitive types of SETL : int, chars, bits, real and atom. An optional range descriptor may be appended to the first three to indicate

minimum and maximum size of the corresponding object. For example :

```
repr X,Y,Z : int(0,1000) ;
     C1,C2 : chars(100) ;
     R : real ;
end ;
```

As this illustrates, the range descriptor part of a primitive mode has the form (n1,n2), where n1 and n2 are integers. The range specifier (0,n2) can be abbreviated (n2).

2.4.2 Bases

The declaration 'base(M)' describes a base whose elements have mode M. Bases must be declared before appearing within other mode descriptors. To give a mode descriptor for base elements is optional.

2.4.3 Derived Modes

Derived modes indicate relationships to a specific base. The only mode descriptor belonging to this category is the member basing mode (e.g., $\in$B), which specifies that the value of a program variable is an element of a base B.

2.4.4 Composite Modes

Mode descriptors for sets, tuples and maps are constructed recursively from other modes, using the following construction :

1) set(M)(SZ)

describes a set whose elements have mode M, and whose expected size is SZ. Here, as elsewhere, the size parameter is optional. If M is a mode of the form ∈B, where B is a base, the attribute keyword 'local' 'remote' or 'sparse' can be used before the keyword 'set'. The altenative notation {M}, which is equivalent to set(M), is allowed.

2) tuple(M)(SZ) describes a tuple whose components have mode M and whose expected size is SZ.

3) tuple(M1,M2,...)

describes a tuple of known length whose components have the specified modes, M1, M2,...,etc. The alternative notation [M1,M2,...], which is equivalent to tuple(M1,M2,...), is also allowed.

4) map(M1)M2

is the mode descriptor for a map from objects of mode M1 to objects of mode M2, i.e., the domain and the range of the map have the mode {M1} and {M2} respectively. If M1 is a mode of the form ∈B, the keywords 'local', 'remote' and 'sparse'

can be used in the same way as that of a domain based set.

5) smap(M1)M2

is the mode descriptor for a single-valued map. M1 and M2 have the same significance as for 'map'.

6) mmap{M1}M2

is the mode descriptor for a multi-valued map, or relation, whose domain is of mode {M1}. Since the set of images of any point X of its domain has mode M2, M2 must be a designator for a set mode.

7) The three previous mode designators are extended to multi-variate maps. For example, we can write

```
repr F : map(∈B,∈B)int ;
     G : mmap(∈B1,∈B2,∈B3)tuple(chars) ;
end ;
```

For variables the mode of whose value may vary during program execution, the mode 'general' can be specified. Whenever an object X of general mode is assigned to a declared Y, the mode of X will be checked dynamically, and its conformity with the mode of Y will be established. If the modes are not compatible, the program terminates immediately. In the reverse case, when Y is assigned to X, X will inherit the mode of Y and no mode conversion is

required.

It is somtimes convenient to bypass the basing mechanism, and to specify that the representation of a given object is to be disjoint from that of any other, i.e., 'unbased'. If the mode 'unbased' is specified for variable X, it means that if in the course of program execution X receives a value whose representation was based then this value will be reconverted to an unbased form before being assigned to X. Note that we also allow variables to be undeclared (rather than unbased). The processor supplies a basing for undeclared variables, consistent with the ways in which such variables are used and assigned.

Program variables may be declared to have multiple modes. This useful extension of the data representation language allows the user to create multiple representations of the same object. For example,

repr S : set(∈B1), ∈B2 ; end repr ;

allows us to describe S as a domain based subset of B1, which is also to be considered as an element of the second base B2. If this declaration is used then the maps based on B1 can be accessed via elements of S, while the maps defined on S can be based on B2. Another example is

repr X : ∈B1, ∈B2, ∈B3 ; end repr ;

which specifies that the object X is to have three simultaneous representations, as elements of each of the bases B1,B2 and B3.

A user can introduce new mode names with the declarative statement :

mode MODENAME : mode-descriptor ;

The MODENAME can then be used subsequently in the program as a valid mode descriptor. The only restriction is that a mode name can be defined at most once and should be different from language reserved words and program variables.

2.5 A Case Study of the Application of Basings

As an example illustrating the use of basings, we present a version of the interval analysis procedure introduced by Cocke and Allen ; cf. Allen[1972], Schaefer[1973]. The original 'unbased' version of this program is taken from Schwartz[1973], pp. 221-223. To ease understandng, we shall first describe the logic of this procedure informally. Then we give basing declarations as well as program code, and analyze the advantages obtained by the use of basings.

2.5.1 Program Logic

We assume a directed program graph G to be given as a set NODES and a function CESOR which maps each ND∈NODES into the set CESOR(ND) of all its successors. One particular member of nodes is assumed to be distinguished as the entry node of G.

An interval in G is a set S of nodes, containing a distinguished node X called the head of S, such that there is no entry into S except through X, and other nodes in S can be reached from X along a path wholly contained in S, and such that when X is removed, S is free of loops (i.e. of closed paths). It is a characteristic property of intervals that their nodes can be enumerated in such an order that, with the exception of branches terminating at the interval head, all branches between nodes of the interval are 'forward' branches, i.e., go from a node S to a node Y having a larger serial number in the enumeration of the interval. Such an enumeration of nodes is said to be an enumeration in interval order. The interval of a node X is the largest interval with X as head ; it may consist of X only. The procedure INTERVAL shown below determines the interval of a node X, and enumerates the nodes of this interval in interval order.

An interval is called maximal if it is not contained

in any larger interval. It can be shown that every program graph can be decomposed uniquely into a union of maximal intervals, and that distinct maximal intervals are disjoint. To find these maximal intervals, we proceed as follows. Take the interval generated by the entry node of the program graph ; this is a first maximal interval. Then take any point X which is a successor of some point Y belonging to an interval already formed, but which does not itself belong to an interval already formed. Form the interval of X ; this is a new maximal interval. The routine INTERVALS given below realizes this process. It also associates, with each maximal interval INT, the set FOLLOW(INT) of all nodes which are successors of a node of INT but do not belong to INT ; and associates with each node B of G the maximal interval INTOV(B) which contains it.

The derived graph G' of a program graph is defined as follows : the nodes of G' are the intervals of G ; the successors of an interval INT are the intervals distinct from it which contains successors of the nodes within INT ; the entry node of G' is the interval containing the entry node of G. The derived graph of G is built up by the routine DG given below.

A program graph in which there exists no interval containing more than one node is called an irreducible

graph ; fortunately, such graphs arise only rarely in connection with actual programs. In SETL, we may write the condition of irreducibility very simply as

(#NODES) = #INTERVALS(NODES).

By forming successive derived graphs G',(G')',etc. of an original graph G, we obtain the derivation sequence of G. In cases in which this sequence converges to a graph consisting only of a single node, the interval-analysis method which has just been outlined gives a decisive account of program flow. In particular, it determines the order in which many other optimization-related processes should be applied to G and to the program P of which G is the flow graph. The derived sequence of G is built up by the main routine given below.

2.5.2 The Interval Analysis Code

Now we present detailed code and basing declarations for the algorithm described above.

```
module      INT-ANALYSIS ;

$ First we declare all global variables.

$ ALLNODES is a base on whch other variables are based.
$ CESOR maps each node and interval into the set of its
$ successor nodes or intervals.
```

```
$ INTOV maps each node into the interval containing it.
$ NODES is the set of nodes in current graph.
$ FOLLOWERS is the set of all nodes which follow some node
$ of an interval I being constructed, but have not yet been
$ added to I.
$ INTS is the set of intervals in current graph.
$ FOLLOW maps each interval I into the nodes which follow
$ a node of I but do not belong to it.
$ ENTRYINT is the entry node of a derived graph.

vars
   ALLNODES, CESOR, INTOV,
   NODES, FOLLOWERS, INTS, FOLLOW, ENTRYINT
end vars ;

$ We then declare based representations of these variables.
repr
   ALLNODES : base  ;
   NODES, INTS : sparse{∈ALLNODES} ;
   CESOR, FOLLOW : local smap(∈ALLNODES)sparse{∈ALLNODES} ;
   INTOV : local smap(∈ALLNODES)∈ALLNODES ;
   FOLLOWERS : sparse{∈ALLNODES} ;
   ENTRYINT : ∈ALLNODES ;
   ENTRY : ∈ALLNODES ;
   SEQ : tuple(tuple(sparse{∈ALLNODES},∈ALLNODES)) ;
end repr ;

$ This is the main routine which constructs the entire
```

```
$ derived sequence of a graph.

L01:  read CESOR ;
L02:  read ENTRY ;
L03:  NODES := {N1: N2:=CESOR(N1)} + {N2: N2:=CESOR(N1)} ;
L04:  SEQ := [[NODES, ENTRY]] ;
L05:  ( while #DG(ENTRY) < #NODE )
L06:     SEQ || [[INTS, ENTRYINT]] ;
L07:     [ NODES, ENTRY ] := [ INTS, ENTRYINT ] ;
      end while ;
      print SEQ ;

proc     DG(ENTRY) ;

$ This routine constructs the derived graph of a graph.

repr
   ENTRY, I : ∈ALLNODES ;
end repr ;

L08:  INTS := INTERVALS(ENTRY) ;
L09:  ENTRYINT := INTOV(ENTRY) ;
L10:  ( ∀ I∈INTS )
L11:     CESOR(I) := INTOV[FOLLOW(I)] ;
      end ∀I ;
      return INTS ;
end proc DG ;

proc    INTERVALS(ENTRY) ;
```

```
$ This routine constructs all intervals of a graph.
$ SEEN is the set of all nodes which are successors of some
$ node in an interval already constructed but which have
$ not themselves been added into any interval.

repr
   ENTRY, NODE, ND : ∈ALLNODES ;
   SEEN, HEADS : remote{∈ALLNODES} ;
   J : tuple(∈ALLNODES) , ∈ALLNODES ;
end repr ;

L12:  INTS := nl ;   FOLLOW := nl ;   INTOV := nl ;
L13:  SEEN := {ENTRY} ;   HEADS := {ENTRY} ;
L14:  ( while SEEN /= nl )
L15:     NODE from SEEN ;
L16:     HEADS with NODE ;
L17:     INTS with ( J := INTERVAL(NODE) ) ;
L18:     FOLLOW(J) := FOLLOWERS ;
L19:     (∀ND := J(N))  INTOV(ND) := J ;   end ∀ND ;
L20:     SEEN := SEEN + ( FOLLOWERS - HEADS) ;
      end while ;
      return INTS ;
end proc INTERVAL ;

proc    INTERVAL(NODE) ;

$ This routine constructs the interval with NODE as
$ the head node.
```

```
repr
   NODE, X, Y, U : ∈ALLNODES ;
   NPREDS, COUNT : local smap(∈ALLNODES)int ;
   NEWIN : sparse{∈ALLNODES} ;
   Z : ∈ALLNODES ;
   INT : tuple(∈ALLNODES) ;
end repr ;

$ Count the number of predecessors of every node.

L21:  (∀X∈NODES) NPREDS(X) := 0;   COUNT(X) := 0;  end ∀X ;
L22:  (∀X∈NODES,Y∈CESOR(X)) NPREDS(Y):=NPREDS(Y)+1 ; end ∀X ;

$ Initialize the interval under construction to be null, and
$ set FOLLOWERS to be {NODE}.

L23:  INT := nult ;
L24:  FOLLOWERS := {NODE} ;

$ Set COUNT(NODE) equal to the number of predecessors of NODE.

L25:  COUNT(NODE) := NPREDS(NODE) ;
L26:  (while NEWIN:={Y∈FOLLOWERS | NPREDS(Y)=COUNT(Y)} /= nl)
L27:     (∀Z∈NEWIN)
L28:        INT || [Z] ;
L29:        FOLLOWERS less Z ;
L30:        (∀U∈CESOR(Z) | U /= NODE )
L31:           COUNT(U) := COUNT(U) + 1 ;
L32:           FOLLOWERS with U ;
            end ∀U ;
```

```
        end ∀Z ;
      end while ;
      return INT ;
end proc INTERVAL ;

end module INT-ANALYSIS ;
```

2.5.3 Efficiency Analysis

We now analyze the effect of the basing choice described above on the execution efficiency of this program.

L01 : Since CESOR is a map based on ALLNODES with images also based on ALLNODES, each component of the elements of CESOR (these elements are all pairs) has to be inserted into ALLNODES if it has not been in ALLNODES yet. This is the typical overhead incurred when we read a based object.

L02 : A base pointer has to be derived for ENTRY after its value is read in.

L09 : No hash-search operation is required in performing this assignment, because INTOV(I) is based on the same base as ENTRY.

L10,L11 : The efficiency of this loop is considerably improved by the basing declarations. Since I is

based on ALLNODES, no hash-search operation is required to reach FOLLOW(I) and CESOR(I). Similarly, no hash-search is required to compute INTOV[FOLLOW(I)].

L12 : Since FOLLOW and INTOV are based maps, it will take slightly longer to initialize their images than to initialize a standard null set.

L18,L19,L21,L22,L23 : No hash-search or conversion operations are required. Note that except for the iteration over J in L19, which utlizes the basing 'tuple(∈ALLNODES)', other occurrences of J use its member basing.

L26 : NEWIN is created slightly more efficiently than if it were unbased. This is because that the hash code of Y is available when it is retrieved from FOLLOWERS ; no hashing is required when Y is inserted into NEWIN.

L29,L32 : No hashing of Z (or U) is necessary ; but search for Z (or U) in the hash table representing FOLLOWERS is still required.

L30 : The inequality test can be done by comparing the basing pointers of U and NODE since both are based on ALLNODES ; considerable advantage is achieved by this, especially when processing derived graphs for which each node is represented by a composite

object.

L31 : No hash-search is required.

In summary, by basing the variables in this program we have eliminated most of the hash-search operations which are otherwise required at run time. The cost we pay for this is that we have to insert, into the base ALLNODES, each interval J created at L17 and also each component of the elements of CESOR read in from the input. Since the cost of inserting an element into a base is roughly the same as the cost of a hash-search opearation, we can estimate the efficiency gain of our basing choice by comparing the frequency of the insertion operations we introduced versus the frequency of the hash-search operations saved. Clearly, the former operations have much lower total frequency than the latter opeations, and therefore our basing choice is indeed advantageous.

2.5.4 Comments on the Foregoing Example

The example presented above shows that by properly selecting the basing of each program variable, consistent basing relations can be defined, eliminating most of the hash-search operations that would otherwise be required, especially in nested loops. Note that we use the term 'consistency of basings' to indicate that no or few

conversions are implied by value assignments. Achieving this property is one of the major issues in devising basings. When suitably high basing consistency is achieved, the efficiency of SETL programs will be improved considerably.

The selection of proper basing modes for program variables is not as simple as one might think. Basing choices cannot be made simply by determining inclusion/membership relations among variables. It will not always be appropriate to base a subset T of a set S on S. As illustration of this, note that in the preceding exmple, FOLLOWERS is a subset of NODES but not based on NODES. Attaining consistency of basings among program variables is the most important issue involved in basing choice. We hope to do this automatically in many cases, but it appears that in some cases consistency can only be achieved through an understanding of program logic. In such cases, some (hopefully quite small) degree of program reconstruction will be required.

CHAPTER 3 : AN AUTOMATIC DATA STRUCTURE CHOICE SYSTEM

The basing declaration language provides a flexible tool which can alleviate much of the work needed to realize an algorithm efficiently. However, a substantial effort is still required to choose good basings. Our long range goal is to remove this burden from users of the SETL system, i.e., to generate good basings without user intervention. We have attacked this goal empirically by taking a representative variety of algorithms and making the basing choices for them manually. By noting the facts which repeatedly appear relevant in this process, we have taken a first essential step toward mechanizing data structure choice. The automatic system which we have constructed can therefore be regarded as an embodiment of various heuristic principles which grew out of our systematized experience of manual basing choice.

3.1 Essential Observations

Based representations provide a systematic mechanism for optimizing set-theoretic operations. The gain they can provide is twofold.

A) If basings are properly chosen, operations on based sets and maps can be performed without hash-table searches. The hashing and clash-list scanning otherwise

required can be replaced by indexing operations.

B) If good basing choices succeed in simplifying SETL operations sufficiently, the code for the remaining SETL primitives can be emitted on-line, eliminating the interpretive overhead imposed by the calls to off-line hash-table accessing procedures.

The main costs involved in using based representations are also twofold :

A) When based objects are generated, their bases are built simultaneously behind the scenes. Inserting a new element into a set forces its parallel insertion into the corresponding base, an operation slightly more expensive than normal (unbased) set insertion, because an element block, generally several words long, must be allocated.

B) Bases are bulky : each element block must accomodate the value of all based functions that are defined on some subsets of the base. If the domains of these functions cover only a small portion of the base, the wasted space can be considerable.

Leaving aside for now the question of storage optimization, it is important to notice that both cost and gains connected with the use of basings can be quantified in terms of the number of insertion operations and hash-searches performed. The generic term 'locate' is

henceforth used to denote this class of operations, i.e., insertion operations and hash-searches. Note that the cost of set insertion is itself the combination of a hash-search and a storage request, and little will be lost if we disregard the latter. This means that we will disregard the difference between an unbased set insertion, and a base insertion. If we use this somewhat simplified measure, the major goal of basing selection can be characterized as that of reducing the number of locate operations required during program execution.

3.2 Fundamental Idea

A locate operation can be avoided at a set or map operation if the arguments of the operation are properly based, i.e., if locate operations have been executed at certain points before the current instruction is reached. Thus the aim of basing choice may be defined as that of moving locate operations from the points at which an object is used to certain program points which follow the point at which the value of the object is created but precede its points of use. Such movement can reduce the number of locates required during execution, and thus increase execution efficiency, if the points at which locate operations are executed have lower frequencies than the instruction at which the object is subject to a set or map operation. In manual use of the basing system, this

kind of movement of implicit locate operations is achieved by imposing proper basings on variable occurrences. Our basing selection must thus aim to uncover basings which have this effect.

With this idea in mind, let us consider the following example.

```
L1:   read NODES ;
L2:   read FATHER ;
L3:   ROOT := nl ;
L4:   (∀ X∈NODES )
L5:       Y := X ;
L6:       ( while FATHER(Y) /= Y )
L7:           Y := FATHER(Y) ;   end;
L8:       ROOT(X) := Y ;
      end ;
L9:   print ROOT ;
```

Plainly, implicit locate operations will be required at instructions L6, L7 and L8 if no basings are used. These locate operations can be avoided by letting all the variables be based on the same base and carrying out locate operations during the read operations at L1 and L2. This basing choice is certainly profitable because L6, L7 and L8 will have substantially higher frequency than L1 and L2. But how can we make such a basing choice

systematically? The outline of a possible approach can be put as follows :

We examine the instructions at which implicit locate operations are required. For each of these instructions, we consider how the locate operations which it contains can be avoided. For exmple, at L6 and L7, no locate operation will be required if FATHER and Y are based on the same base. In order to see how such a common basing might be imposed with least expense, we consider the points at which the values of FATHER and Y are created. FATHER is created at L2 and Y is created at L1 since Y is an element of NODES. Thus we see that if NODES and FATHER are properly based at L1 and L2 then no locate operation will be required at L6 provided X is also based on the same base. This pattern of basings will be profitable because L1 and L2 have lower frequency than L6. We therefore introduce a base B and force NODES and FATHER to be based on B. Repeating the same procedure for L8, we are led to conclude that ROOT and X should be based on the same base. Since X is created at L1 and NODES has been determined to be based on B, we determine that ROOT is also to be based on B. This process associates basings with the objects at L1,L2 and L3 . Using value flow, we can then propagate these basings to other variable occurrences. For example, the X appearing at L4 and L5 is given the basing 'ϵB'. As a final step, we must choose

remote, local or sparse representation for certain composite objects. In the above example, NODES, FATHER and ROOT will all be given local representations since none of them are subject to global operations such as set union or intersection.

This example illustrates our scheme for automatic structure choice :

1. We examine the creation points of the values which appear as arguments to operations for which implicit locate operations are required.

2. We determine proper basings for the values created at the points which have been found in step 1.

3. We propagate these basings to other variables by using value flow.

4. Finally, we determine whether the composite objects which have been based are to have local, remote or sparse representation.

3.3 Overview of the System

We shall now begin to outline an automatic data structure choice system which rests on the fundamental idea explained above. The whole system consists of five distinct phases.

Phase I introduces a base for each 'live period' of a set or map. A 'live period' is used here to mean a set of occurrences of a given variable, which are linked by the data flow maps FFROM and BFROM. The purpose of this phase is to simplify the processing needed in the subsequent phases.

Phase II merges the bases created in the preceding phase, by equivalencing all imputed bases attached to ivariables of a single instruction. In addition, this phase emits the base insertion operations which enforce the postulated basing relations of composite objects. These insertion operations are generated by examining all operations involving hashing ('with', map storage, etc) and by declaring the incorporated item as being an element of the corresponding base. For example, an appearance of X in 'S with X' forces X to be member based on the base B on which S is domain based.

Phase III optimizes the placement of the 'locate' instructions generated in previous steps. This amounts to performing a type of forward code motion on these instructions. The need for such code motion is clear from the following fragment :

$$(\forall X \in S)\ Y = Y + 1\ ;;\ \ldots\ldots\ ;\ Z := F(Y)\ ;$$

The appearance of Y in a map retrieval operation suggests that Y should be an element of the domain base of F. A

'locate' instruction placed at the point of creation of Y will however generate a number of useless basing pointers because all of them (corresponding to successive values of Y in the loop) except the last one are dead (i.e., redefined before they are used). Phase III moves locate instructions out of such loops, and places them at the lower frequency program points where they are actually needed.

Phase IV builds up a detailed description of the mode of each variable occurrence. At this point, all useful basing pointers will have been created at inserted locate instructions. Every variable use which needs basing pointers can count on receiving them without having to execute any locate operations, as long as basing pointers are properly propagated. A base on which only one composite object is domain based is regarded useless (since basing cannot reduce the number of locate operations required during execution) and will therefore be dropped. This phase mainly determines how these basing pointers should be propagated.

Finally, phase V refines our basing choices, by selecting local, remote or sparse representations for based sets and maps .

After this general introduction, we now begin to

explain the detailed workings of each phase of our data structure choice system.

3.4 Phase I : Base Generation

This is a preparatory phase, which generates a base for each 'live period' of a set or map. A 'live period' is used here to mean a set of occurrences of a given variable, which are linked by the chaining functions FFROM and BFROM. The purpose of this phase is to simplify the processing needed in the subsquent phases.

In order to understand the purpose of this phase, let us review the fundamental idea of our algorithm. Our intent is to examine the creation points of the values of the arguments of set and map operations, determine the basings of the ovariables at these creation points and then propagate these basings to other variable occurrences. A difficulty in directly implementing this idea is that the basing propagation process may be somewhat complicated and inefficient. Since at an occurrence of a composite object we may need to know member basing pointers as well as domain basing and/or type information, quite complex information may have to be propagated. This can cause the basing propagation process to be even more inefficient than the type finder algorithm. Another difficulty is caused by the fact that

the type finding algorithm in the SETL optimizer may determine that the types of certain variable occurrences are indefinite, e.g., set-or-map. Due to the incompatability between the based representations of different types of composite objects, based representations are unsuitable for the composite objects of indefinite gross type. We to give such objects the standard representations. Moreover, objects of this kind can never carry basing pointers for their elements. This fact makes it necessary to revise the ideal rule specifying exclusive use of creation points to generate basing pointers, since the basing pointers generated at creation points may be lost during the path to set or map operations. This is illustrated by the following example :

```
L1 : Y := Y + 1 ;

L2 : S with Y ;

L3 : Z from S ;

L4 : U := F(Z) ;
```

Suppose that F is a map and S is of indefinite type. In this case, we should not generate basing pointers at L1 even though Y is a creation point of the Z appearing at L4. Since S has standard representation, the Z appearing

at L3 can never be based in a way which will eliminate an explicit locate operation, even though the Y inserted into S at L2 is based. Thus we will need to generate basing pointers at L3 after a value has been retrieved from S.

In order to overcome these difficulties, we slightly modify our fundamental idea by defining 'live periods' of composite objects and using pseudo creation points instead of creation points. A 'live period' is defined here to be a set of occurrences of a given variable, which are linked by the data flow maps FFROM and BFROM. We treat the domain basings of composite objects differently from member basings. For each live period of a composite object, if all the occurrences in it are of the same gross type, we introduce a base as its domain base. However, no bases are introduced for the live periods which consist of occurrences of indefinite gross types. Then at each set or map operation we impose the condition that the base of the set or map be the member base of the element objects which appear in the instruction. This information is propagated to all of the pseudo creation points of these element objects. A pseudo creation point of an occurrence X is an occurrence which is the ovariable of a value creation or value retrieval instruction and whose value can be transmitted to X through simple assignment instructions. All member basings transmitted to the same pseudo creation point are then identified and explicit

locate instructions are emitted. However, when a pseudo creation point X is the ovariable of a value retrieval instruction in which the composite object Y from which a value is to be retieved is domain based, the basing mode of X is identified with the element mode of Y, but no locate instruction is generated, since X will inherit a basing point from Y. Finally, the member basings postulated at pseudo creation points are propagated through the program, and the bases of different composite objects are suitably merged.

The basing propagation process in this modified approach remains straightforward in view of the fact that only member basings, instead of complete basings involving complex type information, are propagated. The use of pseudo creation points instead of creation points solves the second difficulty mentioned above. Values retrieved from composite objects of indefinite gross type are treated as pseudo creation points for which explicit locate instruction are generated, if necessary. Values retrieved from composite objects of definite gross type are assumed to inherit basing pointers, and therefore no locate instructions are generated.

It should be pointed out that the introduction of a base for each live period of a set or map implicitly implies that structure conversion will never be required

for the occurrences of the same variable. This is because two variable occurrences of composite objects which may transmit values from one to the other or both can transmit or receive values from a third variable occurrence will always have the same basings. We believe that even manual data structure choice will rarely require the type of conversion that our system forbids and therefore that only a very little amount of data structuring power will be lost due to this constraint. It should also be noted that the reason we introduce a basing for each live period of a variable having a composite value, rather than simply associating a basing for each variable name, is that a variable might be used for different purposes at different points in a program.

To facilitate the adjustment of modes during phase V, it is convenient to assume that tuples are also based, i.e. that their components are elements of some base set. Introduction of such bases is harmless, because if no composite objects end up being based on them, they will be dropped during subsequent phases.

3.5 Phase II : Locate Emission and Base Equivalencing

This phase is central to our system. It secures enforcement of the basings chosen in phase I, by generating 'base insertion' ('locate') instructions for

all variable occurrences whose values might be incorporated into a composite object. For example, the instruction :

```
S1 := S with X;
```

leads to the basing relation :

```
X : ∈B ;
```

where B is the base previously assigned to the variable occurrence of S. This basing relation for X is enforced by emitting 'locate' instructions for the ovariable occurrences belonging to the set PS-CRTHIS{X}. Here, PS-CRTHIS{X} is the set of pseudo creation points of X, i.e., occurrences which are the ovariables of value creation or value retrieval instructions and whose values can be trasmitted to X through simple assignment instructions.

A similar approach is taken to map retrieval and store operations. If in phase I the map F has been assigned the mode 'map(∈B1)∈B2', then the instruction

```
F(X) := Y ;
```

will imply the basing relation

```
X : ∈B1 ;
Y : ∈B2 ;
```

In this case, locate instructions (into B1 and B2) are emitted for the occurrences in PS-CRTHIS{X} and PS-CRTHIS{Y}, respectively.

Note that these 'locate' instructions are not directly inserted into the code, but are kept in a temporary set, for the following reasons :

A) The bases being used at this stage are not the actual bases which will appear at run-time. Actual bases will be determined subsequently by building up equivalence classes of the base names introduced in phase I.

B) Some bases may eventually prove useless, because they support only one composite object, in which case all 'locate' instructions which reference them must be dropped.

As we proceed in enforcing basing relations, equivalence relations emerge among bases. Suppose that we are in the process of generating a locate instruction to insert the value appearing at a pseudo creation Y into the base B1 of X. Then if Y is the ovariable of a value retrieval instruction and the composite object from which the value is to be retrieved has been domain based on a base B2 (i.e., the composite object is of an unambiguous type and a base has been introduced for it during the phase I) so that Y can be expected to be member based on

B2, then we just equivalence the bases B1 and B2 without generating any locate instruction. Moreover, even if Y cannot be expected to be member based but Y has already been assigned a locate instruction which will insert its value into a base B3, we do not generate a new locate instruction either, but just equivalence the bases B1 and B3. Certain other instructions force similar base equivalencing rather than generating locate instructions : e.g., set union and intersection force their arguments have the same base.

The existence of an equivalence relation between two bases B1 and B2 means that B1 and B2 are to be considered as two names of the same actual base B, (which will emerge as the representative of the equivalence class to which B1 and B2 belong) . The base equivalencing procedure is carried out by using the compressed balanced tree technique described by Hopcroft and Tarjan. Equivalence classes of bases are represented by a forest of trees. The root of each tree in this forest is the representative (and is called the real base) of the bases in the tree. Trees are structured using a map PARENT ; PARENT(B) points to the parent node of B in the tree containing B if B is not a root, otherwise PARENT(B) is undefined.

The process of base equivalencing and locate generation just described is complicated by the existence

of procedure calls and the need to take variable and base scopes into account. For a given variable occurrence VO, for which a base BO has been suggested, the following may be the case :

A) VO is an occurrence of a global variable v. Then it is reasonable to assign to all its occurrences the same basing (or more precisely, to associate one global base with each of its live periods. See above). The base associated with such variables is therefore a global base.

B) VO is an occurrence of a formal parameter of the procedure P. Then if a base exists for VO, this base is a formal one ; each call to P will instantiate it, by passing to P some actual base AB, (which will be the base of the actual calling parameter AV, to which VO corresponds). It is then reasonable to require that all actual parameters at various points of call have the same form as that chosen for VO, but the actual bases in each case may be distinct and it would be unwise to equivalence them (since equivalencing more bases than strictly necessary may lead to the creation of very sparse objects). But it is reasonable to equivalence all the bases which may appear at a given point of call. This is achieved by partitioning PS-CRTHIS(VO) according to the point of call by which a given occurrence VOX becomes the value of VO. Then the bases occurring in each such partition can be equivalenced. The following example

illustrates this idea.

```
proc A ;                          proc B ;
   .                                 .
X1 := {U1,...} ;                  X2 := {U2,...} ;
Y1 := {V1,...} ;                  Y2 := {V2,...} ;
C(X1,Y1) ;                        C(X2,Y2) ;
   .                                 .
end proc A ;                      end proc B ;

                  proc C(X,Y) ;
                     .
                  X + Y ;
                     .
                  end proc C ;
```

In this case, the bases of X1 and Y1 are equivalenced, and the bases of X2 and Y2 are equivalenced but the bases of X1 and X2 are not equivalenced.

Note that if VO is not a formal parameter, but is nevertheless linked to the formal parameters of P through value-flow, then the preceeding remarks still apply : VO may be based on a formal base, i.e. some base of the formal parameters of P. In such cases, the same partitioning of PS-CRTHIS according to points of call is used.

C) Finally, VO may be local to P , i.e. it may be a local variable whose value is created only within P, and which

does not enter into any operation whose other argument are global or linked to points of call of P. In that case, V0 (and the other arguments of operations in which V0 appears), receives an actual local base.

3.6 Phase III : Locate Insertion

This phase moves 'locate' instructions out of loops. This motion is performed whenever the basing pointer generated by a 'locate' insruction is not actually used within the loop. The following case is typical : a variable X is known to be '$\in$B' and PS-CRTHIS(X) includes the occurrences of X shown in the following text :

```
(∀I := 1...100) X := X + Y ;;
```

Phase II will have hypothesized a locate instruction initially taken to lie within the loop for the ovariable X occuring therein. However, it is clearly unwise to actually put this locate instruction within the loop, because none of the values assigned to X (except the last one) is used as a base element : the basing pointer is dead within the loop. The proper place for the locate instruction is, of course, the exit from the loop. Generally speaking, a locate instruction can be moved out of an interval if no use is made within the interval of the basing pointer which it generates. This can be ascertained by following FFROM of the (previously) located

variable. If we reach an operation which uses the basing pointer within the interval then the locate instruction cannot be moved. If the use appears in some successor interval then it is advantageous to move the locate operation to the head of that interval.

The following procedure systemizes the process of locate instruction motion. We scan the FFROM chain for each occurrence OI at which a locate instruction has been suggested in phase II. The scanning procedure continues until we find all the places at which the basing pointer created at OI might potentially be used. The intervals which contain these points are called the target intervals of OI, and a map MOVETO summarizing this information is generated. If one of the target intervals of OI is the interval in which OI resides, MOVETO(OI) is defined as nl. We use MOVETO to insert actual locate instructions as follows. If MOVETO(OI) is nl then a locate operation is inserted right after OI is created. Otherwise, for each interval INT in MOVETO(OI), a locate operation is inserted at the entry to the largest interval which includes INT but not OI.

Note that the procedure just described is costly but can have significant advantages in some cases. Nevertheless, a study of examples seems to indicate that there are normally a very limited number of bases existing

in a program and very few of them whose associated locate instructions have to be moved. Overall, we judge that this locate movement procedure is marginal and might be omitted if a period of experiment tests conforms the judgement just stated.

3.7 Phase IV : Mode Determination

This phase completes building of the mode descriptor for each variable occurrence.

It is important to note that a base is useful only if at least two composite objects are based on it, because then the basing pointers held by one can be used to access the other. If a base is simply the domain of a map (and nothing else) then nothing is gained by its existence, because there is no way to generate elements of that domain without recalculating the corresponding basing pointer. The same is true if the only objects supported by a base are a set and its elements. In this case, the map (or set) should be unbased.

In this phase, we also determine whether member basing and/or domain basing should be associated with the values appearing at each variable occurrence. It is possible that the values appearing at certain program points shoud be given both member basing and domain basing, due to the

subsequent pattern of uses of the value. In this case, multiple representations are constructed. This is illustrated by the following example :

```
(1)     ( ∀ S∈U )
(2)         (while...)
(3)             (∀X∈S)
                    ....;
(4)                 Y := F(X) ;
                    ....;
                end ∀ ;
(5)             if S in T then....;
            end while ;
        end ∀ ;
```

The S in the first instruction should have a dual basing because the member basing of S is useful in the fifth instruction and the domain basing of S is also useful at the third instruction as the X which retrieves value from S is subsequently subject to a map value retrieval operation at the fourth instruction. Accordingly, S is assigned multiple representations, which have member basing and domain basing, respectively.

This idea is implemented in three steps.

A) For composite objects and member based objects, we replace the member basings referencing dropped bases by element mode of the dropped bases. For example, if (in

pahse I) a set S is tentatively doamin based on a base B1 whose elements are seen (in phase II) to have the mode [∈B2,∈B2] and if B1 is subsequently dropped (because it supports only single composite object) then S is re-assigned the mode 'set([∈B2,∈B2])'.

B) At each occurrence we determine whether domain basings and/or member basings are useful by examining the subsequent uses of the value appearing at this occurrence.

C) We then propagate member basings, starting from locate and value retrieval instructions, to other occurrences which need basings. The propagation procedure ensures that proper basings are carried along with variable values.

3.8 Phase V : Refinement

This phase refines the basing selection made by phase I-IV, by associating the representation attributes, 'local', 'remote' or 'sparse' with set and map representations. The manner in which this is done reflects characteristics of the different representation sructures implied by these keywords.

The advantage of local representation of a map over remote representation lies in the fact that reference to a locally stored map or set is somewhat faster than

reference to a remotely stored map or set, since at least one level of indirection, and probably also an out-of-bounds check, can be avoided. On the other hand, an object having local representation cannot be shared but must be copied at every point at which its share bit would be set. Boolean operations such as =, +, etc., are also relatively inefficient for the locally stored sets, compared with the same operations on objects having the remote set representations.

Another significant point concerns the iteration operator in its relation to based representations. It is clear that the linked hash-table used to represent unbased sets supports iterations efficiently. Iterations over sets having based representations, whether local or remote, are more expensive, because they involve an iteration over the base, and a series of tests for membership in the based subset. The overhead incurred by iterating over the base will be greater the sparser the based object is. It is therefore necessary to review and possibly to revise our primary basing selection for the objects which are subject to iteration operations.

Moreover, proper choice among these three representations depends not only on the kind of use made of the based objects but also the frequencies of these uses and the size of the base. This is not something that

can be discerned statically and thus is information not available from SETL optimizer. In the absence of such information, we adopt a very conservative approach. The heuristics we apply amount to the following :

A) A based object should be sparse if it is to be iterated upon, unless we can show that the object is actually identical in value with its base.

B) If no iteration over an object is performed, but it is an argument to boolean operations (union, intersection, etc) or is passed as a parameter, or is to be copied, or inserted into a larger object, then it should be given remote representation.

C) If only differential updating operations are applied to an object, and it is never transmitted to another by assignment, insertion or call, then it can have a local representation.

3.9 Supplementary Remarks

Our main idea is to insert elements into a base B or to locate elements in B along low frequency paths and carry along the basing pointers thus generated when values are transmitted. This makes it possible to avoid locate operations in high frequency regions. Our technique is therefore a variant of code motion. However, the motion

of locate operations is somewhat different from the motion of more general expressions. To handle more general expressions, we must deal with value preservation issues ; while to handle locate operations we have to deal with somewhat different issue of pointer preservation. The basic idea in moving a general expression EXP is to calculate the value of EXP beforehand and then to use such value without recalculation at the subsequent appearances of EXP. This idea is applicable only if the value of EXP will not be changed between its calculation and its applications. Therefore, a general expression EXP can only be moved without affecting the logic of the program to which EXP belongs within a region in which none of the arguments of EXP are re-defined. On the other hand, we can move a locate operation by inserting a value V into a base to derive a basing pointer and then can avoid re-executing locate operations by utlizing the basing pointer at subsequent applications of V. A variable X which will only hold such 'based' values can therefore be referenced without further locate operations, as long as the basing pointers of 'based' values are transmitted as well as the values. Thus, motion of locate operations is possible under weaker restrictions than code motion of general expressions.

Unlike a manual basing declaration which assigns each declared variable one or more basings, our automatic

basing choice system assigns each variable occurrence one or more basings. This makes it possible for a variable to have different representations at different program points. The required representation conversions for a variable are indicated by simple assignment instructions from the variable to itself of which the ivariables have the representation structures to be converted from and the ovariables have the representation structures to be converted to. This scheme can very often eliminate the necessity to assign a variable multiple representations throughout the whole program or to manually rename some occurrences of a variable and assign a different representation to the newly created variable. In case that a variable does need multiple representations, our algorithm will define the representation which should be referenced at each ivariable occurrence and the representation which should be updated at each ovariable occurrence.

CHAPTER 4 : EXAMPLES

In this chapter, we illustrate the results potentially obtainable by our automatic data structure choice system by examining the way in which it would apply to a number of programs written in SETL. Both the actions and the results of our system are presented. Type information and value flow chains are assumed to be available from the SETL optimizer.

To avoid the special problems which would otherwise be connected with input operations, we insert a set or tuple former instruction after each 'read' statement, to make the structure and the elements of the objects explicit. For example, if the F of the statement 'read F' is known to be of type 'map', we replace the original read statement by the following two instructions.

```
read F' ;
F := { [EF1 := EF(1), EF2 := EF(2) ], EF∈F' } ;
```

Then we let F' to be unbased and try to choose proper representation structure for F. Here, the assignments to EF1 and EF2 are treated as value creations instead of value retrievals. Although certain expansion instructions of this type could be inserted by the SETL optimizer as a result of information obtained by 'backward type analysis', we insert this code exlicitly in the following

examples in order to simplify our discussion. Note also that the necessary insertions can be deduced from 'repr' declarations, if we agree that such a declaration must be given for every variable appearing in a 'read' statement.

As our automatic basing choice algorithm deals with variable occurrences and assigns each of them one or more representation structures, it is impractical to describe the effect of each phase of our algorithm on every variable occurrence. To overcome this expository difficulty, we will use each variable name to represent all of its occurrences unless specified. The basing choice resulting from our algorithm will then be described by an equivalent manual declaration.

In presenting our examples, we first briefly describe the programs to be discussed and list the SETL code analyzed in each case. Then the action of our automatic data structure choice algorithm is tracked phase by phase ; the result expected from each phase is outlined. The final structure choice expected from our algorithm is summarized at the end of each example.

4.1 Example 1 : Tree Traversal

As a first example, we study the postorder tree traversal algorithm given by Knuth[1968]. The inputs to

this algorithm are the root of a tree and two maps defining left and right links respectively. The function of the algorithm is to give each node of the tree its ordinal number in post order.

```
      module TREE-TRAVEL ;

L1:   read ROOT,ILLINK,IRLINK ;
L2:   LLINK := { [LK1:=LK(1), LK2:=LK(2)]: LK∈ILLINK } ;
L3:   RLINK := { [RK1:=RK(1), RK2:=RK(2)]: RK∈IRLINK } ;
L4:   POSTORDER := nl ;          $ Initialize the map POSTORDER.
L5:   STACK := [ ] ;             $ The stack is represented by
                                 $ a tuple.
L6:   NODE := ROOT ;
L7:   ORDINAL := 0 ;
L8:   GO-ON := TRUE ;

L9:   (while GO-ON )
L10:      ( while NODE/=ΩM )
L11:          STACK|||[NODE] ;      $ Push NODE into STACK.
L12:          NODE := LLINK(NODE) ; $ Get left descendant.
          end while ;

L13:      if STACK=[ ] then GO-ON := FALSE ;
                                 $ Check whether stack is empty.
          else                   $ If not,
L14:          NODE := STACK(#STACK) ;
                                 $ Pop out a node form STACK.
L15:          STACK := STACK(1:#STACK-1) ;
```

```
L16:          ORDINAL := ORDINAL + 1 ;
L17:          POSTORDER(NODE) := ORDINAL ;
                                  $ Assign ordinal order to NODE.
L18:          NODE := RLINK(NODE) ;     $ Get right descendant.
          end if ;

      end while ;

L19:  print POSTORDER ;

      end module TREE-TRAVEL ;
```

Four composite objects occurring in this algorithm have to be processed : LLINK and RLINK are input maps, POSTORDER is the output map and STACK is a work stack represented by a tuple. Phase I introduces domain and range bases for these objects.

```
LLINK : map(∈B1)∈B2 ;
RLINK : map(∈B3)∈B4 ;
STACK : tuple(∈B5) ;
POSTORDER : map(∈B6)∈B7 ;
```

In phase II, we examine the set, map and tuple operations at L2,L3,L11 L12,L14,L17 and L18. The pseudo creation points of the occurrence NODE appearing at these instructions are the NODE at L12, L14 and L18, and the ROOT at L1. L2 and L3 suggest that the LK1, LK2, RK1 and RK2 are to be elements of the bases B1, B2, B3 and B4, respectively. Other instructions contribute as follows.

L11 and L14 equivalences B2, B4 and B5 ; L12 equvalences B1, B2 and B4 ; L17 equivalences B2, B4 and B6 ; L8 equivalences B2, B3 and B4. The set of bases is therefore partitioned into two equivalence classes {B1,B2,B3,B4,B5,B6} and {B7}. Let the base name B represent the first equivalence class. The base B7 is dropped since only the range of POSTORDER is based on it.

Phase III physically inserts locate instructions at L1, L2 and L3 ; these put ROOT, LK1, LK2, RK1 and RK2 into the base B. In phase IV, the references to B7, ∈B7, are replaced by the mode of the element of B7, which is known to be integer. The basings of composite objects then become

```
LLINK : map(∈B)∈B ;
RLINK : map(∈B)∈B ;
STACK : tuple(∈B) ;
POSTORDER : map(∈B)int ;
```

Phase IV also determines the following basings for other occurrences.

```
ROOT, NODE : ∈B ;
ORDINAL : int ;
GO¬ON : bool ;
```

Finally, phase V decides that all the composite objects should have local representations.

4.2 Example 2 : Spanning Tree

The second example we present here is the spanning tree algorithm given by Low[1974]. This program computes a spanning tree for a graph. The graph consists of a set of nodes (NODES) and a set of undirected edges between pairs of nodes (EDGES). The program assumes that there is a path (through 0 or more other nodes) between every pair of nodes. A spanning tree for the graph consists of a subgraph containing all the original nodes and a subset of the edges of the original graph such that :

1) For any pair of distinct nodes there exists a unique path between the nodes.

2) There is no path from a node to itself (the subgraph is cycle- free).

```
      module SPANTREE ;

      vars  NODES, EDGES, FATHER  end vars ;

L1 :  read INODES, IEDGES ;
L2 :  NODES := { ND: ND∈INODES } ;
L3 :  EDGES:={ ED:=[ED1:=EDG(1),ED2:=EDG(2)]: EDG∈IEDGES } ;

L4 :  (∀ X∈NODES )              $ Initialize the map FATHER.
L5 :      FATHER(X) := X ; ;

L6 :  ( ∀ E∈EDGES )
```

```
L7 :        F := E(1) ;          $ F and S are the two end nodes
L8 :        S := E(2) ;
L9 :        FG := GROUPOF(F) ;   $ FG and SG are the roots of th
L10 :       SG := GROUPOF(S) ;   $ trees which contain F and S,
                                 $ respectively.
L11 :       if FG /= SG then     $ If F and S are not in the sam
                                 $ tree then
L12 :           TREESET with E ;
L13 :           MERGE(FG,SG) ;   $ their corresponding trees
                                 $ are merged.
            end if ;
        end ∀ ;
L14 : print TREESET ;

        proc GROUPOF(NODE) ;     $ To find the root of the tree
                                 $ which contains node.
L15 : while ( FATHER(NODE) /= NODE )
L16 :       NODE := FATHER(NODE) ;
        end while ;
L17 : return NODE ;

        end proc GROUPOF ;

        proc MERGE(G1,G2) ;      $ To merge two trees.
L18 : FATHER(G2) := G1 ;
L19 : return ;
```

```
    end proc MERGE ;

    end module SPANTREE ;
```

In this module, we deal with six composite objects : NODES is a set, EDGES and TREESET are sets of pairs, FATHER is a single-valued map, and ED and E are tuples of length 2. In phase I, we introduce bases for these variables.

```
NODES : set(∈B1) ;
EDGES : set(∈B2) ;
TREESET : set(∈B3) ;
FATHER : smap(∈B4)∈B5 ;
ED : tuple(∈B6,∈B7) ;
E : tuple(∈B8,∈B9) ;
```

In phase II, we examine all set, map and tuple operations. The information contributed by each instruction is as follows.

L2 : ND is identified as an element of B1.

L3 : ED1, ED2 and ED are identified as elements of B6, B7 and B2 respectively. The mode of B2 becomes 'base([∈B6,∈B7])'.

L5 : The pseudo creation point of X is the X at L4 ; this retrieves X from NODES which is a set based on B1. The bases B1, B4 and B5 are therefore

identified.

L7, L8 : The pseudo creation point of E is the E at L6 which is the ovariable of a value retrieval instruction. The mode of E is identified with the element mode of the base of EDGES. Since B2 already has the mode 'base([∈B6,∈B7])', B8 and B9 are identified with B6 and B7 repectively.

L12 : The pseudo creation point of E is its occurrence at L6. The mode of E is identified with the element mode of the base of EDGES, i.e., B2 and B3 are equivalenced.

L15,L16 : The pseudo creation points of NODE are the F at L7, the S at L8 and the NODE at L16. At L7 and L8, the base B4 is equivalenced with B8 and B9. At L16, the base B4 is equivalenced with B5.

L18 : This instruction has the same effect as the instructions L15 and L16 since the pseudo creation points of G1 and G2 are the same as those of NODE.

The equivalence relations just mentioned lead us to identify the bases B1, B4, B5, B6, B7, B8 and B9 with each other, and the bases B2 and B3 with each other. Furthermore, B2 is known to be a base of elements

[∈B1,∈B1]. Phase III inserts explicit locate operations for ND, ED1, ED2 and ED at L2 and L3. Phase IV propagates the basings that have been derived to other variables. This yields the following basings :

E : ∈B2, [∈B1,∈B1] ; (at L6)

E : ∈B2 ; (at L12)

F, S, NODE, G1, G2 :∈B1 ;

Note that the tuple E at L6 is assigned the member basing '∈B2'in addition to the basing 'tuple(∈B1,∈B1)' which has been assigned to E during phase I, i.e., E is given multiple based representations. The occurrence of E appearing at L7 and L8 will make use of the domain basing and the E appearing at L12 will make use of the member basing.

Finally, noting that all the elements in B2 (i.e., ED at L3) have been inserted into the based subset EDGES of B2, we conclude that EDGES and B2 are identical (in value). Therefore, phase V decides that EDGES as well as other maps and sets (except NODES) can be locally based, even though EDGES is subject to iteration operations. NODES is however given sparse structure because it is subject to iteration operation and we cannot identify it with its base.

The basings which result from these choices can be

summarized by an equivalent declaration as follows.

```
B1 : base ;
B2 : base([∈B1,∈B1]) ;
NODES : sparse set(∈B1) ;
EDGES, TREESET : local set(∈B2) ;
FATHER: local smap(∈B1)∈B1 ;
ED : tuple(∈B1,∈B1) ;
E : ∈B2, tuple(∈B1,∈B1) ;
F, S, NODE, G1 ,G2 : ∈B1 ;
```

4.3 Example 3 : Huffman Coding

The third example to be discussed is Huffman's data compaction algorithm. This algorithm assigns a binary code to each character in such a way that most probable characters receive short codes, while less probable characters receive longer codes. Huffman's technique is as follows : for a given set of characters, CHARS, which we assume to be given along with their expected frequencies, FREQ, take the two characters C1 and C2 of smallest frequency, and hang them as left and right branches from a newly created node N, whose heuristtic meaning is 'either C1 or C2'. Then we remove C1 and C2 from the set of characters and insert N, taking its frequency to be the sum of that of C1 and C2. Repeat this operation until only a single character remains. This

process will grow a tree, the so-called Huffman tree of the set of characters. The code for a character is then its address as a twig of this tree, where 'go down to the left' is represented by a binary 0, and 'go down to the right' is represented by a binary 1.

A code much like the following can be found in Schwartz[1973], pp.148-151.

```
      module HUFFMAN ;

      vars  WORK, WFREQ, L, R  end vars  ;

L1 :  read CHARS, FREQ ;
L2 :  WORK := { CHAR: CHAR∈CHARS } ;   $ Initialize workfile.
L3 :  WFREQ := { [WF1:=WF(1), WF2:=WF(2)]: WF∈FREQ } ;
L4 :  L := nl ;  R := nl ; $ L and R are maps from a node to
                           $ its left and right descendants.

L5 :  ( while #WORK > 1 )
L6 :       WORK less ( C1 := GETMIN(WORK) ) ;
L7 :       WORK less ( C2 := GETMIN(WORK) ) ;
                           $ Get the two nodes with minimal
                           $ frequencies.
L8 :       WORK with ( N := NEWAT ) ;$ Generate a new node.
L9 :       L(N) := C1 ;       $ Build a subtree.
L10 :      R(N) := C2 ;
L11 :      WFREQ(N) := WFREQ(C1) + WFREQ(C2) ;
                           $ Define the frequency of the new
```

```
                                $ node to be the sum of the
                                $ frequencies of its descendants.
      end while ;

L12 : CODE := nl ;
L13 : SEQ := NULB ;
L14 : WALK( TOP := arb WORK ) ;

L15 : print CODE ;


      proc GETMIN(SET) ;
                          $ Get the node with minimal frequency.

L16 : [ KEEP,LEAST ] := [ Y := arb SET , WFREQ(Y) ] ;
L17 : ( ∀ X∈SET )
L18 :     if WFREQ(X)<LEAST then [KEEP,LEAST]:=[X,WFREQ(X)] ;
      end ∀X ;
L20 : return KEEP ;

      end proc GETMIN ;


      proc WALK(T) ;
                                $ To generate code for each node.

L21 : if L(T) /= OM then
L22 :     SEQ || FALSE ;    $ FALSE corresponds to left path.
L23 :     WALK(L(T)) ;
L24 :     SEQ || TRUE ;     $ TRUE corresponds to right path.
L25 :     WALK(R(T)) ;
      else
```

```
L26 :      CODE(T) := SEQ ;
       end if ;
L27 : SEQ := SEQ(1:#SEQ-1) ;
       end proc WALK ;
       end module HUFFMAN ;
```

The composite objects to be discussed in this example include two sets (WORK, SET) and four maps (WFREQ, L, R, CODE). In phase I, we introduce bases for these variables.

```
WORK : set(∈B1) ;
SET : set(∈B2) ;
WFREQ : smap(∈B3)∈B4 ;
L : smap(∈B5)∈B6 ;
R : smap(∈B7)∈B8 ;
CODE : smap(∈B9)∈B10 ;
```

In phase II, we examine the set and map operations appearing in this algorithm. The information contributed by the various set and map instructions is as follows :

L2 : CHAR is identified as an element of B1.

L3 : WF1 and WF2 are identified as elements of B3 and B4 respectively.

L6,L7 : Since the pseudo creation point of both C1

and C2 is X at L17, the base B1 is identified with the base B2.

L8 : N is identified as an element of B1, and a locate instruction is emitted.

L9,L10 : The pseudo creation point of N is the N at L8. Since a locate instruction has already been generated for N at L8, the bases B5 and B7 are equivalenced with B1. The pseudo creation point of C1 and C2 is the X at L17 and hence B6 and B8 are equivalenced with B2.

L11 : B3 is equivalenced with B1.

L16,L18 : B3 is equivalenced with B2.

L21,L23,L25 : B5 and B7 are equivalenced with B1.

L26 : Since the pseudo creation point of T is the TOP at L14, B9 is equivalenced with B1. Since the pseudo creation points of SEQ are the SEQ at L13 and at L27, potential locate instructions are generated for these points.

This leads us to realize that the bases B1,B2,B3,B5,B6,B7,B8 and B9 are all equivalent. Provisional locate instructions are generated at L1, L2 and at L8, at which the insertion of a new atom into WORK also forces the insertion of the same element into the base B1. The bases B4 and B10 can now be dropped because

they only support the range of WFREQ and CODE respectively. The range modes of WFREQ and CODE are seen to be elementary and appear as

```
WFREQ : smap(∈B1)real ;
CODE : smap(∈B1)bits ;
```

Phase III actually inserts the locate instructions noted above. Phase IV propagates the basings assigned to the fundamental objects CHAR, WF1, WF2 and N, and derives the modes of all other variables, yielding

```
C1, C2, N, TOP, KEEP, X, T, NODE : ∈B1 ;
LEAST : real ;
SEQ : bits ;
```

Finally, phase V decides that WFREQ, L, R, and CODE should be locally based, and that WORK and SET should be sparse sets since SET is iterated over in L17.

The data structure choice resulting from our algorithm can be summarized by an equivalent declaration as follows.

```
B : base ;
WORK, set : sparse set(∈B) ;
WFREQ : local smap(∈B)real ;
L, R : local smap(∈B)∈B ;
CODE : local smap(∈B) bits ;
C1, C2, N, TOP, KEEP, X, Y, T, NODE : ∈B ;
```

```
    LEAST : real ;
    SEQ : bits ;
```

4.4 Example 4 : Maximum Flow

Our fourth example is the well-known algorithm for maximum flow through a network, justified by the so-called max-flow min-cut theorem :

Given a network defined by a set of capacities C(P,Q), and given two points X and Y in the network, the maximum value which any flow F from X to Y can have is at the same time the minmum value of the expression

```
[+: C(P,Q), P∈S,Q∈S' ]
```

where S ranges over all sets containing X but not Y, and S' is the complement of S.

The routine takes as arguments a graph defined by set of pairs called GRB and a real-valued capacity function FCAP defined for [P,Q]∈GRB, and two distinct nodes X and Y. Its function is to find the possible maximum flow from X to Y. For an earlier version of this routine and further discussion concerning it, see Schwartz[1973], pp.119-126.

```
    module MAXFLOW ;
```

```
      vars  GRB, FCAP, GRM, F  end vars ;

      macro START(RE) ; RE(2)(if RE(1) then 1else 2) ;  endm ;

      macro FINISH(RE) ; RE(2)(if RE(1) then 2 else 1) ; endm ;

L1 :  read IGRB, IFCAP, X, Y ;
L2 :  GRB := { IEGR := [EG1:=EG(1), EG2:=EG(2)]:EG∈IGRB } ;
                                  $ graph
L3 :  FCAP := { [EF1:=EF(1), EF2:=EF(2)]: EF∈IFCAP } ;·
                                  $ capacity function
L4 :  GRM:={[E(1),[TRUE,E]]:E∈GRB}+{[E(2),[FALSE,E]]:E∈GRB};
                                  $ Map from each node to directed
                                  $ edges.

L5 :  ( ∀E∈GRB )
L6 :      F(E) := 0. ; ;          $ flow function

L7 :  ( while P:=PATH(X,Y) /= OM )
L8 :      AUXFLOWV := [min : RE∈P ] CAP(RE) ;
L9 :      ( ∀ [TVAL,E]∈P )
L10:        F(E) := F(E) + if TVAL then AUXFLOWV else -AUXFLOWV ;
          end ∀ ;
      end while ;

L11 : print F ;

      proc CAP(RE) ;

L12 : return if RE(1) then FCAP(RE(2))-F(RE(2)) else F(RE(2)) ;
```

```
      end proc CAP ;

      proc PATH(X,Y) ;

L13 : NEW := {X} ;
L14 : SET := {X} ;

L15 : (while NEW /= nl doing NEW := NEWER )
L16 :     NEWER := nl ;          $ new nodes to be processed

L17 :     ( ∀ V∈NEW )            $ Look for next plausible node.
L18 :         ( ∀ RE∈GRM(V) | FINISH(RE) =: U notin SET
                                    and CAP(RE) > 0 )
L19 :             PRE(U) := RE ;
L20 :             if U=Y then quit while ;
L21 :             SET with U ;
L22 :             NEWER with U ;
              end ∀ RE ;
          end ∀ V ;
      end while ;

L23 : if U/=Y then return OM ;

L24 : PTH := nl ;
L25 : PT := Y ;

L26 : (while RE:=PRE(PT) /= OM doing PT:=START(RE) )
L27 :     PTH with RE ;          $ Construct the path.
      end while ;

L28 : return PTH ;
```

```
    end proc PATH ;

    end module MAXFLOW ;
```

In this algorithm there occur six sets (GRB, P, PTH, NEW, NEWER, SET), four maps (GRM, FCAP, F and PRE), and two tuples (E, RE). We introduce bases for them as follows :

```
    GRB   : set(∈B1) ;
    P     : set(∈B2) ;
    PTH   : set(∈B3) ;
    NEW   : set(∈B4) ;
    NEWER : set(∈B5) ;
    SET   : set(∈B6) ;
    GRM   : mmap(∈B7)set(∈B8) ;
    FCAP  : smap(∈B9)∈B10 ;
    F     : smap(∈B11)∈B12 ;
    PRE   : smap(∈B13)∈B14 ;
    E : tuple(∈B15,∈B16) ;
    RE : tuple(∈B17,∈B18) ;
```

Noting the operations at L2, L3, L6, L10, L12, L13, L14, L19, L20, L22, L23, L26 and L27, we equivalence the bases B1, B9, B11, B18 the bases B2, B3, B8, B14, and the bases B4, B5, B6, B7, B13. The bases B10, B12, B15, B16 and B17 are found to be useless and dropped. The basings for composite objects then become

```
GRB : set(∈B1) ;
P, PTH : set(∈B2) ;
NEW, NEWER, set : set(∈B4) ;
GRM : mmap(∈B4)set(∈B2) ;
FCAP, F : smap(∈B4)real ;
PRE : smap(∈B4)∈B2 ;
RE : tuple(bits,∈B1) ;
```

Locate instructions are inserted at L1 to locate X and Y into B4, at L2 to locate EGR into B1 and to locate EG1 and EG2 into B4, at L3 to locate EF1 into B2, and at L4 to locate [TRUE,E] and [FALSE,E] into B2. Since each EGR inserted into B1 has the mode [∈B4,∈B4], the element mode of B1 is found to be [∈B4,∈B4], i.e., we have 'B1:base([∈B4,∈B4])'. Similarly, the mode of B2 is found to be 'base([bits,∈B1])'. The modes of other variables are then determined as follows.

```
X, Y, EG1, EG2, EF1, U, V, PT : ∈B4 ;
E, EGR : ∈B1 ;
RE : ∈B2 ;
AUXFLOWV : real ;
TVAL : bits ;
```

Note that the tuple RE at L18 is assigned the member basing '∈B2' in addition to the basing 'tuple(bit,∈B1)' which has been assigned to RE during phase I. The occurrence of RE at L19 will use the domain basing

representation while the rest of the occurrences of RE will make use of its tuple representation.

Finally, NEW, NEWER, P, PTH and each image of GRM are given sparse structure while other composite objects are given local structure. The final basing choice can then be summarized as follows :

```
B4 : base ;
B1 : base([∈B4,∈B4]) ;
B2 : base([bits,∈B1]) ;
GRB : local set(∈B1) ;
P, PTH : sparse set(∈B2) ;
NEW, NEWER : sparse set(∈B4) ;
SET : local set(∈B4) ;
GRM : local mmap(∈B4) sparse set(∈B2) ;
FCAP, F : local smap(∈B4)real ;
PRE : local smap(∈B4)∈B2 ;
RE : tuple(bits,∈B1), ∈B2 ;
X, Y, EG1, EG2, EF1, U, V, PT : ∈B4 ;
E, EGR : ∈B1 ;
AUXFLOWV : real ;
TVAL : bits ;
```

4.5 Example 5 : Interval Analysis

As a final example, we apply our algorithm to the

interval analysis program dicussed in chapter 2, to see whether it can generate a basing choice that is compatible to the manual basing choice described earlier. To clarify the action of our algorithm in this case, we first list certain pseudo creation points which play important roles during the processing.

ENTRY has its pseudo creation point at L02,

ENRYINT has its pseudo creation point at L09,

J has its pseudo creation points (occurrences of INTS) at L12 and L17,

I has its pseudo creation point at L10,

NODE has its pseudo creation point at L15,

Z has its pseudo creation point at L27,

U has its pseudo creation point at L30.

Now let us proceed to track the action of our data structure choice algorithm. In phase I and II, we introduce bases for composite objects and equivalence the bases by examining set, map and tuple operations and the pseudo creation points of their arguments. After phase II, the resulting basing choice is as follows.

```
NODES, INTS, FOLLOWERS, SEEN, HEADS, NEWIN : set(∈B) ;
INTOV : smap(∈B)∈B ;
CESOR : smap(∈B)set(∈B) ;
FOLLOW : smap(∈B)set(∈B) ;
NPREDS : smap(∈B)int ;
COUNT : smap(∈B)int ;
```

```
J : tuple(∈B) ;
SEQ : tuple(set(∈B),∈B) ;
```

Phase III inserts locate instructions at L1 and L2 for the elements of CESOR and ENTRY. Also, a locate operation is inserted at L17 when J receives a value from the routine INTERVAL. Phase IV then propagates the member basings derived from such locate operations to other variables, and yields the followings :

```
ENTRY, ENTRYINT, I, NODE, ND, X, Y, Z, J : ∈B ;
```

At this point the tuple J which has been assigned the mode 'tuple(∈B)' is also assigned the member basing '∈B'. The analysis performed by phase IV indicates that the member basing is useful for all the occurrences of J except the occurrence at L19 which needs domain basing.

The final phase then determines representation attributes in the manner already described, leading to the following overall basing declaration :

```
B : base ;
NODES, INTS, FOLLOWERS, NEWIN : sparse set(∈B) ;
SEEN, HEADS : remote set(∈B) ;
INTOV : local smap(∈B)∈B ;
CESOR, FOLLOW : local smap(∈B)sparse set(∈B) ;
NPREDS, COUNT : local smap(∈B)int ;
J : tuple(∈B), ∈B ;
```

```
SEQ : tuple(tuple(sparse set(∈B),∈B)) ;
```

Encouragingly, the result we have just derived exactly matches the manual choice which was presented in chapter 2 with B corresponding to ALLNODES.

CHAPTER 5 : CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

Basings incorporate pointer and indexing mechanisms and reflect relations between the objects appearing in an program. Using this notion, we have explored a demonstration system which we expect will be capable of automating significant aspects of the data structure choice process. Judging from various test examples, some of which have been presented in the preceding chapter, our system should perform well ; in general it seems to produce a highly acceptable basing choice.

However, the system we have described is far from complete. It utilizes only a small set of representation structures. It incorporates certain systemized heuristics drawn from manual exploration but does not use other more sophisticated data structuring techniques. Much more will need to be done in mastering the complicated problem of automatic data structure choice.

Neverthless, we believe that the concept and the system presented in this thesis have realized a first essential step in automating the data structuring process. Further improvement can certainly be achieved by pushing this approach further. We shall now list some of the idea that have occurred to us during our research in this area,

as possible research topics for the future.

5.1 Merging Rule

The base 'merging rule' described above is one of the crucial parts of our system. It has been over-simplified ; suggested basings of a variable occurrence are merged and identified unless they are radically different. Several possible improvements of basing choice algorithm can be realized by modifying the merging rule.

5.1.1 Parallel Member Basings

When more than one member basing is suggested for an ovariable it may not be necessary to identify these basings. Rather, it may be desirable to keep more than one basing with the vairable and to choose the most advantageous basing for use at each appearance of the variable. In particular, this is useful in the case of two composite objects initialized with the same element which then grow seperately and disjointly. For example,

```
S1 := {X} ;
S2 := {X} ;
(∀0<Y<10) S1 with Y ;
(∀10<Y<20) S2 with Y ;
```

The current system would give S1 and S2 the same basing ;

this is certainly a poor choice because X is the only element which is in both S1 and S2 so that a common base for them would be used sparsely. A better choice is to let X carry two different basings and let S1 and S2 based on different objects.

5.1.2 Small Object Transmission

If we replace the first two instructions of the preceding example by an equivalent pair of instructions

```
S1 := {X} ;
S2 := S1 ;
```

a different consideration arises. In this case, our present automatic structure choice system again proceeds to identify the bases of S1 and S2. However, this is not what we would like to have, since S1 and S2 overlap on X only. A possible solution to this problem would be to treat the assignment instruction 'S2:=S1' as a potential point of conversion, in other words, to treat it as if it read

```
S2 := {Y,Y∈S1} ;
```

which would make it a value creation operation. In this case, the bases of S1 and S2 would not be identified.

However, in an approach like this it is not clear when we should treat an assignment instruction as a potential

conversion and perform such a transformation . A reasonable heuristic rule would be to perform this transformation when the value to be transmitted is known to be a 'small object'. By definition, a small object is a null set or a composite object which is created by a set former consisting of a finite number of explicitly listed elements. A variable is a small-object variable if its value is known to be a small-object value. This definition allows 'small-object' to be regarded as a static attribute of program variables such that a standard attribute propagation algorithm can be applied to detect these cases.

5.2 Conversion of Representation Structure

Domain based objects are given unique representations by the current basing selection system. On the other hand, it is clear from examples that appropriately inserted representation conversion can achieve more efficient execution. An important case is that in which the uses of a variable in different regions of a program suggest different representation attributes. It might be profitable to convert variable representations at 'bottlenecks' between regions within which different representation attributes are suggested.

5.3 Conversion of Basings

This is an even more complicated issue than conversion of representation structure. It can be advantageous to convert the basing of a based object, e.g., from being based on one base to another base. or from being based to being unbased. It is, however, unclear when this kind of conversion will be most profitable and how to detect situations in which this kind of conversion is profitable at all.

5.4 Conversion Of Sparse objects

We treat based objects over which iterations are executed as potential sparse sets. Another possible solution to the problem of how to handle iterations efficiently is to convert the based set to have a list structure before it is iterated over. Such a scheme can be profitable if the necessary conversion can be moved out of frequently executed loops. After conversion is performed, two different representation structures of the same value exist, and the list structure can support the iteration efficiently. However, if a value being iterated over is modified after conversion but before iteration, both representations of the value must be updated, adding to the expense of the scheme. A reasonable compromise might be as follows. A conditional conversion is inserted at the last point preceding the iteration over an object which modifies the object. The density of the object (the cardinality of the object and its base) is examined. If

the density is less than certain value (i.e., if the object is sparse), a conversion is carried out. Since the object is not subject to any modification before it is iterated over, updating of the list created to support iteration will not be required.

5.5 Multi-level Basings

The basing system allows a declared base U to be based on another base V. In such case, the base U is called an intermediate base, and the associated relational structure is said to involve multi-level basings. Conversely, if there is no intermediate base in a relational structure it is said to involve only simple basings.

While our system can only generate simple basings, multi-level basings can be useful, particularly for sparse sets. The introduction of an intermediate base B1 (a base of elements of another base B) can make a based set which is sparse in the ground base B be dense on B1. Change of the density of based sets can significantly improve the efficiency of algebraic operations as well as iteration over based sets. However, extra locate operations are required whenever an element of the intermediate base references the ground base for the first time (after the first such reference, a proper basing pointer can be kept with the element in the intermediate base). The cost of element block allocation for the intermediate base can

also increase significantly.

5.6 Co-linked Bases

Use of co-linked bases may be regarded as a variation of multi-level basing. Two bases are co-linked if each one is declared to be a base of elements of the other. An example is given by the legal declaration

```
repr  B1 : base(∈B2) ;  B2 : base(∈B1) ;  end repr ;
```

After this declaration each element of B1 keeps a pointer to the corresponding element of B2 and vice versa. A basing pointer from an element X1 in B1 to the corresponding element X2 in B2 is established at the first reference from X1 to X2 and vice versa. After such a pointer is established succeeding references from X1 to X2 need no additional locate operations. If X1 never references the corresponding element of B2, a pointer from X1 of B1 to the corresponding element of B2 need never be made available. Clearly, co-linked basing is more general than strict multi-level basing in the sense that either base can be regarded as an intermediate base of the other

CHAPTER 6 : SETL CODE FOR THE DATA STRUCTURE CHOICE ALGORITHM

```
     module AUTO-DSTRUCT ;

$      Our data structure choice algorithm has been designed
$ to be compatible with the currently implemented SETL
$ optimizer, i.e., analytic information derived by the
$ SETL optimizer can directly be used by our algorithm.
$ For a detailed account of the SETL optimizer, see
$ Grand[1978].  The terms, variable names and data
$ structures used in the SETL optimizer are inherited by our
$ algorithm.  Some of the utility macros and subroutines
$ defined in the SETL optimizer are also used by our
$ algorithm without modification.

$      For this reason, we shall now summarize the
$ definitions, constructs and ouputs of the SETL optimizer
$ which are relevant to our subsequent discussion.

$ Symbols

$      Each symbol corresponds to a resolved name in a SETL
$ source program or to a compiler generated temporary.
$ Symbols are represented as atoms which are elements of
$ the base SYMBOLS.

$ The Symbol Table
```

```
$     The 'symbol table' is a collection of maps on SYMBOLS.
$ These maps are:

     vars
          NAME,          $ name of symbol
          VALUE,         $ value of symbol
          IS-CONST,      $ indicates constant
          IS-BASE,       $ indicates base
          IS-GLOBE       $ indicates global variable
     end vars ;

$ Program

$     A program is divided into routines, basic blocks, and
$ instructions.  Each instruction consists of an opcode and
$ a tuple of arguments.  All the inputs and outputs of an
$ instruction appear explicitly as arguments.

$     The instructions in each block are threaded into a
$ linked list.  This is designed to allow maximum
$ flexibility in code insertion and deletion.

$ Maps on Instructions

     vars
          NEXT,          $ next instruction in block
          BLOCKOF,       $ gives block containing instruction
          OPCODE,        $ operation code
          ARGS,          $ tuple of arguments
```

```
        COPY-FLAG     $ indicates what copy action should
                      $ be done
    end vars ;

$ Macros for Accessing Fields within Instructions

(M1):   macro ARG1(I);  ARGS(I)(1)   endm;  $ 1St argument
(M2):   macro ARG2(I);  ARGS(I)(2)   endm;  $ 2nd argument
(M3):   macro ARG3(I);  ARGS(I)(3)   endm;  $ 3rd argument

$ Iteration over a Program

$     The following macro is used to iterate over the
$ instructions in a block.

(M4):   macro FORALLCODE(B, I);
            init I:=FIRST(B); while I/=OM step I:=NEXT(I);
        endm;

$ Iteration over the whole program is written :

$    (∀ B ∈ BLOCKS, FORALLCODE(B, I))

$ Occurrences

$     An occurrence is a use or definition of a variable.
$ It is defined to be a pair

$     [instruction identifier, argument number].

$ Occurences which are inputs are called 'ivariables', and
```

```
$ occurrences which are outputs are called 'ovariables'.  An
$ occurrence may be both an i- and o-variable, for example
$ 'F' in 'F(X) := Y'.

$     With the exception of the 'from' operator, ovariables
$ are always the first argument of their instruction.
$ Ivariables may appear in any argument position.

$     The 'from' operator has two arguments, both of which
$ are inputs and outputs.

$     In order to speed up iterations and test the types of
$ occurrences we provide the following sets and macros :

      vars
         ALL-OI,      $ set of all occurrences
         ALL-O,       $ set of all ovariables
         ALL-I,       $ set of all ivariables
         ALL-VARS     $ set of all entries in symbol table
      end vars ;


$ The following macros are used in connection with
$ occurrences :

(M5):   macro INSTNO(OI);          OI(1)                endm;
              $ the instruction which contains OI

(M6):   macro ARGNO(OI);           OI(2)                endm;
              $ the argument number of OI
```

```
(M7):   macro OI-OP(OI);          OPCODE(OI(1))        endm;
              $ the operation code of the instruction
              $ containing OI

(M8):   macro OI-NAME(OI);        ARGS(OI(1)) (OI(2)) endm;
               $ the variable name of OI

(M9):   macro OI-VALUE(OI);       VALUE(OI-NAME(OI))  endm;
              $ the value of OI

(M10):   macro OI-INTOV(OI) ;
              INTOV(BLOCKOF(INSTNO(OI)))    endm ;
              $ the interval which contains OI

(M11):   macro IS-OVAR(OI);   (OI in ALL-O)  endm;
              $ indicates whether OI is an ovariable

(M12):   macro IS-IVAR(OI);   (OI in ALL-I)  endm;
              $ indicates whether OI is an ivariable

(M13):   macro IS-HASHED(OI) ;  OI-OP(OI) in OPS-HASH  endm;
              $ indicates whether OI is subject to an
              $ operation involving hashing

(M14):   macro IFROMO(O, N);  [O(1), N+1]  endm ;
              $ the N-th ivariable of the instruction
              $ containing O as the ovairalbe

(M15):   macro OFROMI(I);  [I(1), 1]  endm ;
              $ the ovariable of the instruction containing
              $ the ivariable I
```

```
$ Opcodes

$     The set OPCODES defines all the operations in the
$ internal program representation.  The following lists
$ the opcodes relevant to the subsequent discussion.

     const OPCODES :=

          (
$ Binary operators

               Q1~ADD,        $  +
               Q1~DIV,        $  /
               Q1~EXP,        $  **
               Q1~EQ,         $ eq
               Q1~IMP,        $ imp
               Q1~IN,         $ in
               Q1~INCS,       $ incs
               Q1~less,       $ less
               Q1~lessF,      $ lessf
               Q1~MOD,        $  //
               Q1~MULT,       $  *
               Q1~NE,         $ ne
               Q1~notin,      $ notin
               Q1~NPOW,       $ npow(n,set)
               Q1~SUB,        $  -
               Q1~SUBSET,     $ subset
               Q1~with,       $ with

$ Unary operators
```

```
        Q1~UMIN,      $ unary minus

$ Miscellaneous

        Q1~SET,       $ set former
        Q1~SET1,      $ set formed with loop
        Q1~TUP,       $ tuple former
        Q1~TUP1,      $ tuple formed with loop
        Q1~FROM,      $ A1 from A2;

$ Iterators

        Q1~NEXT,      $ A1 := next element of A2
        Q1~NEXTD,     $ A1 := next element of domain A2
        Q1~INEXT,     $ initialize next loop
        Q1~INEXTD,    $ initialize nextd loop

$ Mappings

        Q1~OF,        $ A1 := A2(A3)
        Q1~OFA,       $ A1 = A2{A3}
        Q1~OFB,       $ A1 = A2[A3]
        Q1~SOF,       $ arg1(arg2)=arg3
        Q1~SOFA,      $ arg1{arg2}=arg3
        Q1~SOFB,      $ A1[A2]:= A3;

$ Assignments - all assign arg2 to arg1

        Q1~ARGIN,     $ assign argument to formal parameter
        Q1~ARGOUT,    $ return value from a function
        Q1~ASN,       $ arg1 := arg2
```

```
        Q1~PUSH,      $ push element for set former
        Q1~POP,       $ pop read only argument from stack

       };

    end const;

$ The opcodes are divided into several categories which are
$ represented as constant sets.  These sets are used to
$ drive 'case' statements and as predicates on OPCODES.
$ The classes useful in our subsequent discussion include

    const

(C1):   OPS~ASN :=    $ assignment operators
      { Q1~ASN, Q1~ARGIN, Q1~ARGOUT, Q1~PUSH, Q1~POP } ;

(C2):   OPS~HASH :=    $ operations involving hashing
      { Q1~WITH, Q1~LESS, Q1~FROM, Q1~OF, Q1~OFA } ;

(C3):   OPS~RETRIEVE :=    $ value retrieval operators
      { Q1~ARB, Q1~FROM, Q1~NEXT, Q1~INEXT, Q1~NEXTD,
        Q1~NEXTD, Q1~INEXTD, Q1~OF, Q1~OFA, Q1~OFB } ;

(C4):   OPS~CREATE :=  $ value creation operators
      { Q1~ADD, Q1~DIV, Q1~EXP, Q1~LESS, Q1~LESSF, Q1~MOD,
        Q1~MULT, Q1~NPOW, Q1~SUB, Q1~WITH, Q1~UMIN, Q1~SET,
        Q1~SET1, Q1~TUP, Q1~TUP1 ,Q1~SOF, Q1~SOFA,
        Q1~SOFB } ;
    end const;
```

$ RC-strings

$ In order to make interprocedural analysis precise, we
$ allow the attributes of a variable occurrence OI to vary
$ depending upon how the routine P in which OI appears is
$ invoked. For example, X may be of type 'set' when P is
$ called form an instruction A, but of type 'tuple' when P
$ is called from another instruction B. For this reason
$ most of the attribute maps (e.g., type) used within the
$ SETL optimizer are defined on pairs of [OI,RC] where OI is
$ a variable occurrence and RC is a so-called RC-string
$ (R-eturn C-all string), instead of being defined simply
$ on variable occurrences. Logically, a RC-string is the
$ concatenation of a series of return-call phrases.

$ Each return-call phrase is a pair

$ [XXX, INST]

$ where XXX is one of the constants RC-CALL or RC-RETN, and
$ INST is an instruction identifier. Intutively,
$ [RC-CALL,I] means 'by way of CALL at instruction I' and
$ [RC-RETN,I] means 'by way of return to instruction I'.
$ An entire RC string is represented as a tuple of these
$ pairs.

$ Technically, when we analyze an attrbibute of an
$ occurrence OI we store it as a pair (or set of pairs)
$ [RCS,ATT] where ATT is the attribute and RCS is the
$ return-call path along which the attribute was created.

```
$ When we propagate an attribute [RCS,ATT] from an
$ occurrence OI to some other occurrence I later in the
$ program, we must begin by finding the return-call path
$ RCS1 which takes us from OI to I.  We then set the
$ attribute of I to the pair [ATT,RCS||RCS1] if this
$ concatenation yields a valid RC-string and to OM
$ otherwise.

$     The operator 'A cc. B' returns the concatenation of
$ two RC-strings A and B if the result is a valid string
$ and returns the constant ERROR-PATH otherwise.  For
$ further detail about RC-strings, see Grand et al[1978].

$ The following constants are used for RC-strings :

     const
(C5):     RC-CALL;    $ indicates CALL
(C6):     RC-RETN;    $ indicates return
(C7):     NULL-PATH := [ ];  $ null return CALL path
(C8):     ERROR-PATH;    $ error path
     end const;

$ Chaining Of occurrences

$     Certain central algorithms in the SETL optimizer are
$ designed to build up a set of pairs [OI,I] where OI and
$ I are occurrences and there is a path with certain
$ properties from one to the other.  We say that these
$ sets 'link' or 'chain' occurrences with certain
```

```
$ properties.

$     When we link two occurrences in different procedures,
$ we keep track of the return-call path by which they are
$ linked.

$     The sets built by the various chaining algorithms
$ always have elements of the form:

$     [OI, [P, I] ]

$ where OI and I are occurrences linked together along the
$ return- call path P.

$     Three of the most important link maps are
$ BFROM[I] :

$     If I is an occurrence of some variable V then BFROM[I]
$ is a set of pairs [P,OI] where P is a return-call path
$ and OI is an occurrence of V such that there is a
$ V-clear path along P from OI to I.

$ FFROM[I] :

$     This is essentially the inverse of BFROM.  If I is an
$ occurrence of V then FFROM[I] is a set of pairs [P,OI]
$ where P is a return-call path and OI is an occurrence of
$ V such that there is a V-clear path along P from I to
$ OI.

$ PS-CRTHIS[I] :
```

```
$     This is the value flow map used in our data structure
$ choice algorithm.  If I is a variable occurrence then
$ PS-CRTHIS{I} is a set of pairs [P,OI] where P is a
$ return-call path and OI is the ovariable occurrence of a
$ value creation or value retrieval instruction such that
$ the value of OI can be transmitted to I through simple
$ assignments (along the path P).

$ Type Finding

$     The SETL optimzer uses a modified version of the
$ Tennenbaum's type finder.  This type finding algorithm
$ is interprocedural in nature.  The type information it
$ develops gives us a first approximation to the
$ representation structure of each occurrence.

$     The set of basic types form a Boolean lattice.  A
$ point on this lattice is referred to as a gross type.
$ Intuitively the gross type of an object gives us
$ information about its top level structure.

$     The type lattice is defined in terms of a set of
$ nodes, namely the gross types, and a MEET and JOIN
$ function.  The gross types are represented as sets of
$ atoms, and MEET and JOIN are represented as set union
$ and intersection.

 const
(C9):     TOM    := { NEWAT };  $ OM
```

```
(C10):   TSI   := { NEWAT };  $ short integer
(C11):   TLI   := { NEWAT };  $ long integer
(C12):   TR    := { NEWAT };  $ real
(C13):   TSC   := { NEWAT };  $ short chars
(C14):   TLC   := { NEWAT };  $ long chars(C15):
(C15):   TA    := { NEWAT };  $ atom
(C16):   TL    := { NEWAT };  $ label
(C17):   TP    := { NEWAT };  $ procedure
(C18):   KNT   := { NEWAT };  $ known length tuple
(C19):   UNT   := { NEWAT };  $ unknown length tuple
(C20):   GMAP  := { NEWAT };  $ map
(C21):   GSET  := { NEWAT };  $ set
(C22):   TELMT := { NEWAT};   $ element
 end const;

$ The following points in the type lattice are also given
$ names:

 const
(C23):   TC   := TSC + TLC;        $ characters
(C24):   TI   := TSI + TLI;        $ integers
(C25):   TNUM := TI  + TR;         $ numbers
(C26):   TMTUP := KNT;             $ tuples of know length
(C27):   THTUP := UNT;             $ tuples of unknown length
(C28):   TTUP := KNT + UNT;        $ tuples
(C29):   TSET := GSET;             $ sets
(C30):   TMAP := GMAP;             $ maps

(C31):   TG := TOM + TA + TNUM + TC + TTUP + TSET + TMAP;
```

```
(C32):   TZ := nl;                    $ zero element

(C33):   TSTRUCT := TTUP + TSET + TMAP;  $ any structured type
(C34):   TZSTRUCT:= TG - TSTRUCT;  $ zero for sublattice of
                                   $ structures

(C35):   MAPTUP  := TMAP + TTUP;   $ map or tuple
(C36):   SETTUP  := TSET + TTUP;   $ set or tuple
(C37):   MAPSET  := TMAP + TSET;   $ map or set
 end const;

$ The following macros are used to access type lattice
$ elements.

(M16):   macro STRUCTPART(G);   G * TSTRUCT     endm;
              $ struture of type G

(M17):   macro IS-PRIM(G);   STRUCTPART(G) = TZ    endm;
             $ indicate whether G is a primitive type


$     Note that the following two criteria are used to
$ regulate the degree to which minor type ambiguities can
$ impact our data stucture choice algorithm.

$ (1) We assume that the object S appearing in an instruction
$     'S with X' will be assigned the type TMAP because a map
$     cannot be defined on OM and the current type finder is
$     unable to tell that whether the first component of X (if
$     it is a tuple) is OM.

$ (2) We assume that the object T appearing in 'T(X)' will be
```

```
$      assigned the type TMTUP only if the value of X is known
$      at the compile time (i.e., OI¬VALUE(X) is defined),
$      otherwise T should be assigned the type THTUP if it is
$      a tuple.

$ Type Descriptors

$      A type descriptor is a complete description of an
$ object's type.  If an object is primitive, it is
$ described by a pair [GROSSTYP,OM] where GROSSTYP is an
$ element of the type lattice indicating its gross type.
$ If an object is structured, its type is described as a
$ pair [GROSSTYP,COMPTYP] where GROSSTYP is again an element
$ of the type lattice and COMPTYP is a type descriptor for
$ the components of the object.

$      The use of COMPTYP varies slightly for each type of
$ structured object.

$ A. Sets

$     GROSSTYP: TSET
$     COMPTYP:  type descriptor for elements

$ B. Homogeneous tuples of unknown length

$     GROSSTYP: THTUP
$     COMPTYP:  type descriptor for components

$ C. Mixed tuple of known length
```

```
$     GROSSTYP: TMTUP
$     COMPTYP:  tuple of type descriptors for components.

$               If T is a type descriptor for a known length
$               tuple, then CTYPN(T,N) is the type descriptor
$               for the N-th component of T, and LENTYP is the
$               length of the tuple of type descriptors, or
$               equivalently, the length of the tuple.

$ D. Map

$     GROSSTYP: TMAP
$     COMPTYP:  type descriptor for the element type of the map,
$               namely known tuple of length 2.  If T is a type
$               descriptor for a map, then DOMTYP(T) is a type
$               descriptor for the domain of T, and RANTYP(T)
$               is a type descriptor for the range (i.e., F(X))
$               of T.

$      The output of the type finder is a map called TYPES.
$ If OI is an occurrence and P is a return-call path, then
$ TYPES(OI,P) is a type descriptor giving the type of OI,
$ assuming that the program has proceeded along the
$ return-call path P.

$ The following macros are used for type descriptors :

(M18):    macro GROSSTYP(T);      T(1)                endm;
                    $ gross type of type descriptor T
```

```
(M19):   macro COMPTYP(T);       T(2)              endm;
                $ type of elements or components of T

(M20):   macro CTYPN(T, N);      COMPTYP(T)(N)     endm;
                $ type of N-th component of T

(M21):   macro LENTYP(T);        (# COMPTYP(T))    endm;
                $ length of type descriptor T

(M22):   macro DOMTYP(T);  CTYPN(COMPTYP(T), 1)  endm;
                $ domain type of type descriptor T

(M23):   macro RANTYP(T);  CTYPN(COMPTYP(T), 2)  endm;
                $ range type of type descriptor T
```

$ Automatic Data Structure Choice

```
$       Our data structure choice algorithm utlizes the
$ information derived by the SETL optimizer to determine
$ the basing mode of variable occurrences.  The inputs to
$ our algorithm are :

$       1. The data flow maps BFROM and FFROM, and the value
$          flow map PS-CRTHIS.

$       2. The type map TYPES which gives the possible types
$          of each occurrence.

$ The output from our algorithm is a map MODE which maps
$ each variable occurrence into an appropriate 'mode
$ descriptor'.  In addition, 'locate' instructions, which
```

$ are designated by simple assignment instructions with
$ member basing modes for ovariables and non-member basing
$ modes for ivarables, are also inserted into program
$ code.

$ We allow unique representation structure for each
$ variable occurrence OI, regardless how the routine in
$ which OI appears is invoked. Unlike most of the other
$ attribute maps defined in the SETL optimizer, which map
$ variable occurrences to pairs [RC-STRING,ATTRIBUTE], the
$ map MODE will map each variable occurrence into a single
$ mode descriptor.

$ Mode Descriptor

$ A mode descriptor is a complete description of the
$ representation structure of an object. It has a structure
$ similar to that of a type descriptor, but is represented
$ as a tuple of length four instead of a tuple of length
$ two. The detailed structure of a mode descriptor is

$ [GROSSTYP, COMPTYP, BASENAM, REPRATT]

$ The first two fields GROSSTYP and COMPTYP have the same
$ meanings as they have in a type descriptor. However,
$ TEMLT which indicates member basing and TBASE which
$ indicates bases are introduced as new gross types. When
$ the GROSSTYP of a mode descriptor is TELMT, the BASENAM
$ field contains the name of its base. The last field

```
$ REPRATT is used to describe the representation attribute
$ of domain based objects.  The allowed representation
$ attributes are defined by

 const
(C39):    SPARSE    := { NEWAT };  $ sparse representation
(C39):    REMOTE    := { NEWAT };  $ remote representation
(C41):    LOCAL     := { NEWAT };  $ local representation
 end const;

$ Particular examples of mode descriptors are :

$ 1. The mode 'ϵB' is represented as
$      [TELMT, OM, B, OM].

$ 2. The mode 'local set(ϵB)' is represented as
$      [TSET, [TELMT, OM, B, OM], OM, LOCAL].

$ 3. The mode 'sparse smap(ϵB1)ϵB2' is represented as
$      [TMAP,[TMTUP,[[TELMT,OM,B1,OM],[TELMT,OM,B2,OM]],OM,SPARSE]

$ 4. The mode 'base(int)' is represented as
$      [TBASE,TI,OM,OM].

$ Two macros, in addition to those defined on type
$ descriptors, are used to reference mode descriptors.

(M24):    macro BASENAM(M) ;  M(3)   endm ;
                $ base name of mode descriptor M

(M25): macro REPRATT(M) ;  M(4)   endm ;
                $ representation attribute of mode
```

```
                $ descriptor M

$ The following additional macros are used to manipulate
$ mode descriptors of variable occurrences.

(M26): macro MGTYP(VAR) ; GROSSTYP(MODE(VAR)) endm ;
                          $ gross type of a variable
                          $ occurrence

(M27): macro GLTYP(VAR) ; LENTYP(MODE(VAR))  endm ;
                          $ length of a tuple

(M28): macro ELMBASE(SET) ; BASENAM(COMPTYP(MODE(SET))) endm ;
                          $ base of a based set

(M29): macro DOMBASE(MAP) ; BASENAM(DOMTYP(MODE(MAP))) endm ;
                          $ base of the domain of a based
                          $ map

(M30): macro RANBASE(MAP) ; BASENAM(RANTYP(MODE(MAP))) endm ;
                          $ base of the range of a map

(M31): macro COMBASE(TUP,N) ; BASENAM(CTYPN(MODE(TUP),N)) endm
                          $ base of the N-th component
                          $ of a tuple


$ Global Variables

$    In addition to the global variables used in the SETL
$ optimizer, we introduce the following global variables
$ in our algorithm.
```

```
vars

    MODE ,          $ map from occurrences to their mode
                    $ descriptors

    LIVEPDS ,       $ set of live periods of variables
                    $ having composite object values

    IS-FORMAL ,     $ map on bases to indicate that a
                    $ base only supports the formal
                    $ parameters of a procedure

    NBASE ,         $ map on real bases to count the
                    $ number of bases in the same
                    $ equivalence class

    NBASEDON ,      $ map on bases to count the number
                    $ of sets and maps based on them

    PARENT ,        $ map from bases to their preceding
                    $ nodes in the equivalence class tree

    LOCINS ,        $ set of possible 'locate'
                    $ instructions to be inserted

    BASE-ELMTS,     $ set of occurrences which are known
                    $ to be elements of bases

    ID-TO-BASE,     $ map on variable occurences
                    $ indicating whether the occurrence
                    $ values are indentical with their
                    $ bases
```

```
    HASH-USE,          $ map on variable occurrences
                       $ indicating whether the occurrence
                       $ values are subsequently subject to
                       $ operations involving hashing

    NON-HASH-USE       $ map on variable occurrences
                       $ indicating whether the occurrence
                       $ values are subsequently subject to
                       $ operations not involving hashing

end vars ;

$ Useless Bases

$      Our data structure choice algorithm first introduces a
$ base for each composite object and then equivalences
$ bases.  After this equivalencing procedure, some bases
$ may eventually be found useless.  A base is useful only
$ if at least two composite objects are based on it,
$ because then the basing pointers held by one can be used
$ to access the other.  If a base is simply the domain of
$ a map (and nothing else) then nothing is gained by its
$ existence, because there is no way to generate elements
$ of that domain without recalculating the corresponding
$ basing pointer.  The same is true if the only objects
$ supported by a base are a set and its elements.  In this
$ case, the map (or set) should be unbased.  Consequently,
$ any base which supports only a single composite object
```

```
$ is useless, unless the object is a formal parameter of a
$ procedure.  To detect such a case, we provide the
$ following macro.

(M32):    macro CAN-DROP(B) ;
                NBASEDON(B) = 1 and not IS-FORMAL(B)
          endm ;

$ Representations for the Global Variables used

$     We now declare the representations for the global
$ variables used by the SETL optimizer.

 repr

$ Variables defined in the SETL optimizer

(V1):     SYMBOLS : base ;    $ base of symbols
(V2):     NAME : smap(∈SYMBOLS)char ;
                              $ name of a symbol
(V3):     VALUE : smap(∈SYMBOLS)real ;
                              $ value of a symbol
(V4):     IS-CONST : smap(∈SYMBOLS)bool ;
                              $ indicates constant
(V5):     IS-GLOB : smap(∈SYMBOLS)bool ;
                              $ indicates global variable

(V6):     OI-BASE : base ;    $ base of ovariable occurrences
(V7):     RC-BASE : base ;    $ base of RC-strings
(V8):     ALL-OI : set(∈OI-BASE) ;
```

```
                              $ set of all variable
                              $ occurrences
(V9):    ALL-O : set(∈OI-BASE) ;
                              $ set of all ovariable
                              $ occurrences
(V10):   ALL-I : set(∈OI-BASE) ;
                              $ set of all ivariable
                              $ occurrences
(V11):   BFROM : mmap{∈OI-BASE}set([∈RC-BASE,∈OI-BASE]) ;
                              $ data flow map
(V12):   FFROM : mmap{∈OI-BASE}set([∈RC-BASE,∈OI-BASE]) ;
                              $ data flow map
(V13):   PS-CRTHIS : mmap{∈OI-BASE}set([∈RC-BASE,∈OI-BASE]) ;
                              $ value flow map

(V14):   TYPE-BASE : base ;
                              $ base of type descriptors
(V15):   TYPES : mmap{OI-BASE}set([∈CR-BASE,∈TYPE-BASE]) ;
                              $ possible types of variable
                              $ occurrences

(V16):   INSTRS : base(int) ;
                              $ base of instructions
(V17):   BLOCK-BASE : base ;
                              $ base of code blocks and
                              $ intervals
(V18):   OPCODES : base ;     $ base of opcodes
(V19):   NEXT : smap(∈INSTRS)∈INSTR ;
```

```
                              $ next instruction
(V20):    BLOCKOF : smap(∈INSTRS)∈BLOCK¬BASE ;
                              $ code block containing the
                              $ specified instruction
(V21):    OPCODE : smap(∈INSTRS)∈OPCODES ;
                              $ operation code of an
                              $ instruction
(V22):    ARGS : smap(∈INSTRS)tuple(∈OI¬BASE,∈OI¬BASE,∈OI¬BASE)
                              $ arguments of an instruction

$ variables particular to the data structure choice
$ algortihm

(V23):    SB¬BASE : base ;    $ base of all generated bases
(V24):    IS¬FORMAL : smap(∈SB¬BASE)bool ;
                              $ indicates whether a base is a
                              $ formal base
(V25):    NBASES : smap(∈SB¬BASE)int ;
                              $ number of bases in the same
                              $ equivalence class as a geven
                              $ base
(V26):    NBASEDON : smap(∈SB¬BASE)int ;
                              $ number of sets and maps based
                              $ on a base
(V27):    BASE¬ELMT : mmap[∈SB¬BASE]SET(∈OI¬BASE) ;
                              $ occurrences inserted into a
                              $ base

(V28):    MODE¬BASE : base ; $ base of mode descriptors
```

```
(V29):   BMODE : map(∈SB-BASE)∈MODE-BASE ;
                                $ mode of a base
(V30):   MODE : map(∈OI-BASE)∈MODE-BASE ;
                                $ basing modes of an occurrence

(V31):   LPD-BASE : base ;  $ base of live periods
(V32):   LIVEPDS : set(∈LPD-BASE) ;
                                $ set of live periods

(V33):   HASH-USE : smap(∈OI-BASE)bool ;
                                $ map on variable occurrences
                                $ indicating whether the
                                $ occurrence values are
                                $ subsequently subject to
                                $ operations involving hashing
(V34):   NON-HASH-USE : smap(∈OI-BASE)bool ;
                                $ map on variable occurrences
                                $ indicating whether the
                                $ occurrence values are
                                $ subsequently subject to
                                $ operations not involving
                                $ hashing
(V35):   ID-TO-BASE : smap(∈OI-BASE)bool ;
                                $ map on variable occurrences
                                $ indicating whether the
                                $ occurrence values are
                                $ indentical with their bases

 end repr :
```

```
$ Program Organization

$     To make it easier to understand the global structure
$ of our algorithm, we outline its major subroutines in
$ their calling hierarchy.

$ (P1):  GENBASE          - generate bases
$ (P2):    CONSTR-PS-CRTHIS  - construct PS-CRTHIS map
$ (P3):  GENLOCS          - generate locate instructions
$ (P4):    MERGEOBJ       - process set algebraic operations
$ (P5):    PROPELMT       - process set insertion operations
$ (P6):    PROPOFMAP      - process map retrieval operations
$ (P7):    PROPSOFMAP     - process map storage operations
$ (P8):    PROPOFTUP      - process tuple retrieval operations
$ (P9):    PROPSOFTUP     - process tuple storage operations
$ (P10):   PROPSOFAMAP    - process map range storage
$                          - operations
$ (P11):     MERGE        - merge bases
$ (P12):       MERGE-INTO   - merge the mode of an occurrence
$                             - with the element mode of a base
$ (P13):   INSERTLOCS     - insert locate instructions
$ (P14):     EQUIV        - equivalence bases
$ (P15):       REALB      - find real bases
$ (P16):       MODEDIS.   - calculate mode disjunction
$ (P17):     PARTITION    - partition pseudo creation points
$ (P18):       LASTCALL   - find last calling point of
$                           - a procedure
```

```
$ (P19): MOVELOCS       - move locate instructions
$ (P20): UPDMODES       - update occurrence modes
$ (P21):   MODECMPRS    - compress mode descriptors
$ (P22):     SUBSTMD    - substitute mode descriptors
$ (P23):   USE-DETERM   - determine uses of variable values
$ (P24):   BASING-PROP  - propagate basing mode
$ (P25): REFINE         - refine occurrence modes
$ (P26):   ID-BASE      - verify occurrences identical
$                       - with bases
$ (P27):     SETOF      - find sets constructed by set
$                       - formers
$ (P28):   MAKE-REMOTE  - choose remote representations

$ Cross-reference Listing of Global Names

$ For reference purpose, we list in the appendix B all
$ global names used in our algorithm in their alphabetical
$ order.

$ SETL code

$ Now we are ready to present the code.

proc AUTO-DATA public ;

$ This is the main routine of our algorithm.

$ Initialize global variables.
MODE := NBASES := LIVEPDS := LOCINS := NBASEDON := PARENT := nl ;
```

```
BASE¬ELMTS := IS¬TO¬BASE := HASH¬USE := NON¬HASH¬USE := nl ;

GENBASE();                    $ invoke phase I
GENLOCS();                    $ invoke phase II
MOVELOCS();                   $ invoke phase III
UPDMODES();                   $ invoke phase IV
REFINE();                     $ invoke phase V

end AUTO¬DATA ;


proc GENBASE ;

$ The purpose of this procedure is to improve the efficiency
$ of the subsequent phases.

$ This procedure generates a base for each live period of a
$ composite object.  A live period is used here to mean a
$ set of occurrences of a given variable, which are linked
$ by the chaining maps FFROM and BFROM and can therefore
$ be expected to have the same basing.  However, bases are
$ generated only for the live periods in which all the
$ occurrences have the same gross type.  No bases are
$ generated for the bases which consists of occurrences of
$ indefinite gross type (e.g., TSETTUP and TMAPTUP).
$ Objects of indefinite gross type can never be doamin
$ based.  Each base generated in this phase initiate a
$ seperate equivalence class.  Equivalence classes will be
$ merged in phase II.
```

```
$ To facilitate the adjustment of modes during phase V, it
$ is convenient to assume that tuples are also based,
$ i.e., that their components are also elements of some
$ bases.  Introduction of such bases is harmless, because
$ if no composite objects end up being based on them, they
$ will be dropped.

$ This routine is called by the main routine AUTO-DATA and
$ calls the routine CONSTR-PS-CRTHIS to construct
$ PS-CRTHIS map for subsequent use.  This routine also
$ calls a utility routine DIS in the type finder to find
$ the disjunction of a set of type descriptors.

$ The global variables referenced by this routine include
$     LIVEPDS      - set of live periods
$     NBASES(B)    - number of bases in the same equivalence
$                  - class as B
$     NBASEDON(B) - number of sets and maps based on base B
$     IS-FORMAL(B) - indicates whether B is a formal base
$     All-OI       - all variable occurrences
$     MODE(OI)     - mode of occurrence OI
$     BFROM(OI)    - occurrences to which OI is directly
$                  - linked
$     FFROM(OI)    - occurrences which are directly
$                  - linked to OI
$     TYPES(OI)    - possible types of occurrence OI

$ The macros used in this routine include
$     OI-NAME(OI) - name of occurrence OI, see (M8).
```

```
$      GROSSTYP(T) - gross type of type descriptor T, see (M18).
$      LENTYP(T)   - length of type descriptor T, see (M21).
$      COMPTYP(M)  - elemeht mode of mode descriptor M, see (M19
$      DOMTYP(M)   - domain mode of mode descriptor M, see (M22)
$      RANTYP(M)   - range mode of mode descriptor M, see (M23).
$      CTYPN(M,I)  - I-th component mode of mode descriptor M,
$                  - see (M20).

$ The local variables defined in this routine are

 repr
      TODO : set(∈OI¬BASE) ; $ workpile of variable occurrences
      WORK : set(∈OI¬BASE) ; $ workpile of variable occurrences
      TPO,TYP,T : ∈TYPE¬BASE ;   $ type descriptors
      OI,WOI : ∈OI¬BASE ;    $ variable occurrences
      BASE : ∈SB¬BASE ;      $ base
      NEWM : ∈MODE¬BASE ;    $ mode descriptor
      LPD : ∈LPD¬BASE ;      $ live period
      L : int ;              $ length of tuple
 end repr ;

$ Initialize TODO to be the set of all variable occurrences.
TODO := ALL¬OI ;

$ An initial mode descriptor is assigned to each variable
$ occurrence.

( while TODO /= nl )

    OI := arb TODO ;          $ Get an occurrence.
```

```
T := DIS. / { TYP : [-,TYP]∈TYPES(OI) } ;
                        $ Disjunction of all possible
                        $ types of OI.

$ If OI is of primitive type, take its type as the
$ initial mode descriptor.

if IS-PRIM(T) then
    MODE(OI) := T ;
    TODO less OI ;
    continue while TODO ;     $ Process next occurrence.
end if ;

$ Otherwise, OI is a composite object.  Construct the
$ live period containing OI.

WORK := {OI} ;            $ Initialize a workpile.

(while WORK/=nl)

    OI from WORK ;        $ Choose an arbitrary element from
                          $ WORK.
    T := DIS. / { TYP : [-,TYP]∈TYPES(OI) } ;
                          $ Disjunction of all possible
                          $ types of OI.
    MODE(OI) := T ;       $ Use type as initial mode.
    TODO less OI ;        $ OI need not be processed any more.
    LPD with OI ;         $ OI is included into the current
                          $ live period.

    $ Insert the occurrences which are linked to OI and
```

```
    $ have not been processed into the workpile WORK.

    WORK + {WOI : [-,WOI]∈(FFROM{OI}+BFROM{OI}) | WOI in TO

end while WORK ;

$ LPD is a complete live period.
LIVEPDS with LPD ;

T := DIS. / { TYP : [-,TYP]∈TYPES{OI} | OI in LPD } ;
                          $ Disjunction of all possible
                          $ types of the occurrences in
                          $ LPD.

TPO := GROSSTYP(T) ;      $ gross type of T

$ If all the occurrences in the live period LPD have the
$ same definite composite type, construct a domain
$ basing mode for all of them.  The gross type is
$ taken as the initial mode descriptor.  This mode
$ descriptor will be completed subsequently.

NEWM := [TPO] ;           $ template for mode descriptor

case TPO of

(TSET) :

    $ If every occurrence OI in LPD is a set, generate a
    $ base for OI.  Give OI the mode set(∈BASE) by
    $ inserting the member basing ∈BASE into the mode
    $ descriptor NEWM for the elements of OI.  NEWM
```

```
    $ will become [TSET,[TELMT,OM,BASE]].

    COMPTYP(NEWM) := [TELMT, OM, BASE:=NEWAT] ;

    $ Set NBASEDON(BASE) := 1 to indicate that OI is
    $ domain based on BASE.

    NBASEDON(BASE) := 1 ;

    $ Let BASE form an equivalence class.

    NBASES(BASE) := 1 ;

    $ Initialize the mode of BASE.

    BMODE(BASE) := [TBASE,COMPTYP(T)] ;

    $ Initialize BASE to be a formal base.

    IS¬FORMAL(BASE) := TRUE ;

(TMAP) :

    $ If every occurrence OI in LPD is a map, generate
    $ two bases for OI ; one for its domain and the
    $ other for its range.  Give OI the mode
    $ map(∈BASE1)∈BASE2 by inserting the mode
    $ [∈BASE1,∈BASE2] into the mode descriptor NEWM for
    $ the elements of OI.  NEWM will become
    $ [TMAP,[TMTUP,[[TELMT,OM,BASE1],[TELMT,OM,BASE2]]]].

    COMPTYP(NEWM) := [TMTUP,[]] ;
    DOMTYP(NEWM) := [TELMT, OM, BASE1:=NEWAT] ;
```

```
        RANTYP(NEWM) := [TELMT, OM, BASE2:=NEWAT] ;

        $ Set NBASEDON(BASE1) := 1 to indicate that OI is
        $ domain based on BASE1.

        NBASEDON(BASE) := 1 ;

        $ Let BASE1 and BASE2 each form an equivalence
        $ class.

        NBASES(BASE1) := 1 ;
        NBASES(BASE2) := 1 ;

        $ Initialize the mode of BASE1 and BASE2.

        BMODE(BASE1) := [TBASE,DOMTYP(T)] ;
        BMODE(BASE2) := [TBASE,RANTYP(T)] ;

        $ Initialize BASE1 and BASE2 to be formal bases.

        IS-FORMAL(BASE1) := TRUE ;
        IS-FORMAL(BASE2) := TRUE ;

(THTUP) :

        $ If every occurrence OI in LPD is a tuple of unkown
        $ length, generate a base on which all the
        $ components of OI to be based.  Give OI the mode
        $ tuple(∈BASE) by inserting the member basing
        $ ∈BASE into the mode descriptor NEWM for the
        $ components of the tuple.  NEWM will become
        $ [THTUP,[TELMT,OM,BASE]].
```

```
        COMPTYP(NEWM) :=  [TELMT, OM, BASE:=NEWAT] ;

        $ Let BASE form an equivalence class.

        NBASES(BASE) := 1 ;

        $ Initialize the mode of base.

        BMODE(BASE) := [TBASE,COMPTYP(T)] ;

        $ Initialize base to be a formal base.

        IS-FORMAL(BASE) := TRUE ;

(TMTUP) :

        $ If every occurrence OI in LPD is a tuple of known
        $ length, generate a base for each component of OI
        $ and let the component be based on this base.

        L := LENTYP(T) ;     $ length of tuple OI
        COMPTYP(NEWM) := [ ] ;

        (∀I := 1...L)

            $ Generate a base for the component and insert
            $ the member basing ∈BASE into the mode
            $ descriptor NEWM for the component.  NEWM
            $ will eventually become
            $ [TMTUP,[[TELMT,OM,BASE]...]].

            CTYPN(NEWM, I) := [TELMT, OM, BASE := NEWAT] ;

            $ Let BASE form an equivalence class.
```

```
            NBASES(BASE) := 1 ;

            $ Initialize the mode of BASE.

            BMODE(BASE) := [TBASE,CTYPNYP(T,I)] ;

            $ Initialize base to be a formal base.

            IS-FORMAL(BASE) := TRUE
        end ∀I ;

    else

        $ Otherwise, at least one of the occurrences in LPD
        $ is of indefinite gross type.  In this case, no
        $ bases are generated.

        continue while TODO ;
                            $ Process next occurrence.
    end case ;

    $ All occurrences in LPD have the same definite gross
    $ type.  Assign the mode descriptor justed constructed
    $ to all occurrences in LPD.

    (∀OI ∈ LPD)

        MODE(OI) := NEWM ;  $ Assign OI the mode descriptor NEWM.

end ∀ ;

end while TODO ;

$ Call the routine CONSTR-PS-CRTHIS to construct the map
```

```
$ PS-CRTHIS for subsequent use.

CONSTR-PS-CRTHIS() ;

return ;

end proc GENBASE ;


proc CONSTR-PS-CRTHIS ;

$ This routine constructs PS-CRTHIS map.  We start with the
$ ovariables of value creation and value retrieval
$ instructions (these are the total set of pseudo creation
$ points in the whole program), and assign them as the
$ pseudo creation points of themselves.  The pseudo
$ creation map is then propagated through FFROM map and
$ simple assignment instructions ; a pseudo creation point
$ of an occurrence OI must be a pseudo creation point of
$ every occurrence in FFROM{OI} and a pseudo creation
$ point of the ivariable of a simple assignment
$ instruction must be a pseudo creation point of the
$ ovariable of the instruction.

$ The workpile WORK consists of elements having the format

$     [OI, [P,POI]]

$ where POI is a pseudo creation point of OI and P is the
$ path from POI to OI.

$ This routine is called by the routine GENBASE.
```

```
$ The global variables referenced by this routine include
$     BLOCKS        - set of code blocks
$     OPCODE(I)     - operation code of instruction I
$     PS-CRTHIS(OI) - pseudo creation points of OI
$     FFROM(OI)     - occurrences which are directly linked
$                   - to OI

$ The macros used in this routine include
$     FORALLCODE(B,I) - for each instruction I in block B,
$                     - see (M4).
$     IS-IVAR(OI)   - indicates whether OI is an ivariable,
$                   - see (M12).
$     OFROMI(OI)    - the ovariable in the same instruction
$                   - as the ivariable OI, see (M15).

$ The local variables defined in this routine are

 repr
      OVAR,OI,POI,WOI : ∈OI-BASE ; $ variable occurrences
      WORK : set(∈OI-BASE) ;       $ workpile of occurrences
      I : ∈INSTR ;                 $ instruction
      B : ∈BLOCK-BASE ;            $ code block
      P,NP,WP : ∈RC-BASE ;         $ RC-strings
 end repr ;

$ Assign the ovariables of value creation and value
$ retrieval instructions as the pseudo creation points of
$ themselves and insert them into the workpile WORK.
```

```
(∀B∈BLOCKS, FORALLCODE(B,I) |
                    OPCODE(I) in (OPS-CREATE + OPS-RETRIEVE)))

    OVAR := ARG1(I) ;     $ ovariable of the instruction
    PS-CRTHIS{OVAR} := { [NULL-PATH,OVAR] } ;
    WORK with [OVAR, [NULL-PATH,OVAR]] ;

end ∀B ;

$ Process elements in WORK until WORK is emtpy.

(while WORK /= nl)

    $ Retrieve an element from WORK.

    [OI, [P,POI]] from WORK ;

    $ POI is a pseudo creation point of OI and P is the path
    $ from POI to OI.

    $ For each occurrence WOI in FFROM{OI} which can be
    $ reached from POI, POI is a pseudo creation point of
    $ WOI.

    (∀[WP,WOI]∈FFROM{OI} |
                    ( NP := P CC. WP ) /= ERROR-PATH )
                                $ NP is the path from POI to
                                $ WOI.

        $ Insert [NP,POI] into PS-CRTHIS {WOI} if it has not
        $ been inserted in PS-CRTHIS{WOI} yet.

        if [NP,POI] notin PS-CRTHIS{WOI} then
```

```
            PS-CRTHIS(WOI) with [NP,POI] ;
            WORK with [WOI, [NP,POI]] ;
        end if ;
    end ∀ ;
    $ If OI is the ivariable of a simple assignment
    $ instruction the pseudo creation points of OI are
    $ also the pseudo creation points of the ovariable of
    $ the instruction.
    if IS-IVAR(OI) and OI-OP(OI) in OPS-ASN then
        WOI := OFROMI(OI) ;  $ ovariable of the instruction
        PS-CRTHIS(WOI) with [P,POI] ;
        WORK with [WOI, [P,POI]] ;
    end if ;
end while WORK ;
return ;
end proc CONSTR-PS-CRTHIS ;

proc GENLOCS ;
$ This procedure enforces the basings chosen for composite
$ objects, by generating 'base insertion' ('locate')
$ instructions for all variable occurrences whose values
$ might be incorporated into a composite object.  For
$ example, the instruction :
```

```
$     S1 := S with X ;

$ leads to the basing relation :

$     X : ∈B ;

$ where B is the base previously assigned to the variable
$ occurrence of S.  This basing relation for X is enforced
$ by emitting 'locate' instructions for the ovariable
$ occurrences belonging to the set PS-CRTHIS(X) (except in
$ certain cases discussed below).  Here, PS-CRTHIS(X) is
$ the set of pseudo creation points of X, i.e., the
$ occurrences which are the ovariables of value creation or
$ value retrieval instructions and whose values can be
$ trasmitted to X through simple assignment instructions.

$ A similar approach is taken to map retrieval and store
$ operations.  If in phase I the map F has been assigned
$ the mode 'map(∈B1)∈B2', then the instruction

$     F(X) := Y ;

$ will imply the basing relation

$     X : ∈B1 ;     Y : ∈B2 ;

$ In this case, locate instructions (into B1 and B2) are
$ emitted for the occurrences in PS-CRTHIS(X) and
$ PS-CRTHIS(Y), respectively.

$ Note that these 'locate' instructions are not directly
$ inserted into the code, but are kept in a temporary set,
```

```
$ for the following reasons :

$ A) The bases being used at this stage are not the actual
$ bases which will appear at run-time.  Actual bases will
$ be determined subsequently by building up equivalence
$ classes of the base names introduced in phase I.

$ B) Some bases may eventually prove useless, because they
$ support only one composite object, in which case all
$ 'locate' instructions which reference them must be
$ dropped.

$ As we proceed in enforcing basing relations, equivalence
$ relations emerge among bases.  When about to generate a
$ locate instruction to insert a pseudo creation point Y
$ into the base B1 of X, we check to see if Y is the
$ ovariable of a value retrieval instruction and if the
$ composite object S from which Y is retrieved has been
$ domain based on a base B2 (i.e., if S is of a definite
$ gross type and a base has been introduced for it during
$ phase I).  In this case, Y will be member based on B2, and
$ we just equivalence the bases B1 and B2 without
$ generating any locate instruction.  Moreover, if the
$ above condition is not satisfied but if Y has already
$ been assigned a locate instruction which will insert Y
$ into a base B3, we still do not generate a new locate
$ instruction, but just equivalence the bases B1 and B3.
$ Certain other instructions force similar base
$ equivalencing rather than generating locate instructions
```

```
$ : e.g., set union and intersection force their arguments
$ have the same base.

$ If we have equivalenced two bases B1 and B2, then B1 and
$ B2 are considered as two names of the same actual base
$ B, (which will emerge subsequently as the representative
$ of the equivalence class to which B1 and B2 belong) .

$ The process of base equivalencing and locate generation
$ just described is complicated by the existence of
$ procedure calls and the need to take variable and base
$ scopes into account.  For a given variable occurrence
$ V0, for which a base B0 has been suggested, the
$ following may be the case :

$ A) V0 is an occurrence of a global variable V.  Then it is
$ reasonable to assign the same basing to all its
$ occurrences (or more precisely, to associate one global
$ base with each of its live periods.  See above).  The
$ base associated with such variables is therefore called
$ a global base.

$ B) V0 is an occurrence of a formal parameter of the
$ procedure P.  Then if a base exists for V0, this base is
$ a formal one ; each call to P will instantiate it, by
$ passing to P some actual base AB, (which will be the
$ base of the actual calling parameter AV, to which V0
$ corresponds).  It is then reasonable to require that all
$ actual parameters at various points of call have the same
```

```
$ form as that chosen for V0, but in each case we allow
$ the actual bases to be distinct.  It would be unwise to
$ equivalence these bases (since equivalencing more bases
$ than strictly necessary may lead to the creation of very
$ sparse objects), but it is reasonable to equivalence all
$ the bases which may appear at a given point of call.
$ This is achieved by partitioning PS-CRTHIS(V0) according
$ to the points-of-call by which a given occurrence V0X
$ becomes the value of V0.  Then the bases occurring in
$ each such partition can be equivalenced.

$ Note that if V0 is not a formal parameter, but is
$ nevertheless linked to the formal parameters of P
$ through value-flow, then the preceding remarks still
$ apply : V0 may be based on a formal base, i.e. some base
$ of the formal parameters of P.  In such cases, the same
$ partitioning of PS-CRTHIS according to points-of-call is
$ used.

$ C) Finally, V0 may be local to P, i.e., it may be a local
$ variable whose value is created only within P, and which
$ does not enter into any operation whose other arguments
$ are global or linked to points of call of P.  In that
$ case, V0 (and the other arguments of operations in which
$ V0 appears), receives an actual local base.

$ In order to simplify the mode adjustment phase, the
$ arbitrary basings chosen for tuple components and for
$ the range of maps in the preceding phase, are propagated
```

```
$ during the present phase, in the same way as the basings
$ of set elements.  Operations of incorporation, i.e.,
$ tuple assignments, are treated as map stores and the
$ same base equivalencing procedure is used in all cases.

$ Base equivalencing is carried out by using a compressed
$ balanced tree technique.  Equivalence classes of bases
$ are represented by a forest of trees.  The root of each
$ such tree is the representative (and is called the real
$ base) of the bases in the tree.  Trees are structured by
$ map PARENT ; PARENT(B) points to the parent node of B in
$ the tree containing B if B is not a root, otherwise
$ PARENT(B) is undefined.

$ This routine is called by the main routine AUTO-DATA and
$ calls the folowing routines MERGEOBJ, PROPELMT,
$ PROPOFMAP, PROPOFTUP, PROPSOFMAP, PROPSOFTUP and
$ PROPSOFAMAP.  All of these routines perform similar
$ functions, namely generate locate instructions and merge
$ bases, in a manner depending on the operation of the
$ instruction being processed.

$ The global variables referenced by this routine include
$      BLOCKS        - set of code blocks
$      ARGS(I)       - arguments of instruction I
$      OPCODE(I)     - operation code of instruction I
$      PS-CRTHIS{OI} - pseudo creation points of OI

$ The macros used in this routine include
```

```
$      FORALLCODE(B,I)  - for each instruction I in block B,
$                       - see (M4).
$      MGTYP(OI)        - gross type of occurrence OI, see (M26).
$      IS~PRIM(M)       - indicates whether M is a primitive mode,
$                       - see (M17).

$ The local variables defined in this routine are

 repr
      IV1,IV2,OV : ∈OI~BASE ;     $ variable occurrences
      B : ∈BLOCK~BASE ;           $ code block
 end repr ;

$ Iterate through each instruction of the program.

(∀B∈BLOCKS, FORALLCODE(B,I))

    [OV, IV1, IV2] = ARGS(I) ;  $ Unpack instruction

    case OPCODE(I) of

    $ For a comparison operation : equivalence the bases
    $ of the ivariables if they are composite objects.

    (Q1~EQ, Q1~NE, Q1~INCS) :
        if not IS~PRIM(MGTYP(IV1)) then
            MERGEOBJ(IV1,PS~CRTHIS(IV1),IV2,PS~CRTHIS(IV2)) ;
        end if ;

    $ For a simple assignment : equivalence the bases of the
    $ ivariables and the ovariable if they are composite
```

```
$ objects.

(Q1-ASN, Q1-ARGIN, Q1-PUSH, Q1-POP) :
    if not IS-PRIM(MGTYP(IV1)) then
        MERGEOBJ(OV,PS-CRTHIS(OV),IV1,PS-CRTHIS(IV1) ) ;
    end if ;

$ For an algebraic operation : equivalence the bases of
$ the ivariables and the ovariable if they are
$ composite objects.

(Q1-ADD, Q1-SUB, Q1-MULT, Q1-MOD) :
    if not IS-PRIM(MGTYP(IV1)) then
        MERGEOBJ(IV1,PS-CRTHIS(IV1),IV2,PS-CRTHIS(IV2) ) ;
        MERGEOBJ(OV,PS-CRTHIS(OV),IV1,PS-CRTHIS(IV1) ) ;
    end if ;

$ For a set or tuple insertion or deletion operation :
$ equivalence the bases of the ovariable and the first
$ argument, and generate locate instructions for the
$ second argument.

(Q1-WITH, Q1-LESS) :
    if MGTYP(IV1)=TSET or MGTYP(IV1)=THTUP then
        MERGEOBJ(OV,PS-CRTHIS(OV),IV1,PS-CRTHIS(IV1) ) ;
        PROPELMT(IV1,PS-CRTHIS(IV1),IV2,PS-CRTHIS(IV2) ) ;
    end if ;

$ For a membership operation : generate locate
$ instructions for the first argument.
```

```
(Q1-IN, Q1-NOTIN) :
    if MGTYP(IV2)=TSET or MGTYP(IV2)=THTUP then
        PROPELMT(IV2,PS-CRTHIS(IV2),IV1,PS-CRTHIS(IV1)) ;
    end if ;

$ For a set or tuple former : generate locate
$ instructions for each component.

(Q1-SET, Q1-SET1, Q1-TUP, Q1-TUP1) :
    if MGTYP(OV)=TSET or MGTYP(OV)=THTUP then
        PROPELMT(OV, PS-CRTHIS(OV), IV1,CRTHIS(IV1)) ;
    end if ;

$ For a map or tuple retrieval operation : generate
$ locate instructions for the index variable.

(Q1-OF, Q1-OFA, Q1-OFB) :
    if MGTYP(IV1)=TMAP then            $ IV1 is a map.
        PROPOFMAP(IV1,PS-CRTHIS(IV1),IV2,PS-CRTHIS(IV2)) ;
    elseif TTUP incl MGTYP(IV1) then  $ IV1 is a tuple.
        PROPOFTUP(IV1, PS-CRTHIS(IV1), IV2 ) ;
    end if ;

$ For a single-valued storage operation of map or tuple
$ : generate locate instructions for the ivariables.

(Q1-SOF, Q1-LESSF) :
    if MGTYP(IV1)=TMAP then            $ IV1 is a map.
        PROPSOFMAP(OV,PS-CRTHIS(OV),IV1,PS-CRTHIS(IV1),IV2,
                PS-CRTHIS(IV2) ) ;
```

```
        elseif TTUP incl MGTYP(IV1) then  $ IV1 is a tuple.
            PROPSOFTUP(OV,PS-CRTHIS(OV),IV1,IV2,PS-CRTHIS(IV2))
        end if ;

    $ For a multi-valued storage operation : F(X) := S ,
    $ treat the right-hand side differently, and invoke a
    $ seperate routine.

    (Ω1-SOFA) : PROPSOFAMAP(OV,PS-CRTHIS(OV),IV1,PS-CRTHIS(IV1)
                    IV2, PS-CRTHIS(IV2) ) ;

    else                        $ Other opcodes are not examined.
        continue ∀ ;
    end case ;
nd ∀B;

return ;
end proc GENLOCS ;


$ Now follows a family of routines all of which perform
$ similar functions, namely equivalencing bases and
$ inserting locate instructions, but for different kinds
$ of operations.  This family consists of the routines
$ MERGEOBJ, PROPELMT, PROPOFMAP, PROPOFTUP, PROPSOFMAP,
$ PROPSOFTUP and PROPSOFAMAP.  Because of their
$ similarities detailed documentation is provided only in
$ the routine MERGEOBJ.  Please make reference to this
$ routine wherever a lack of documentations is sensed in
$ the other routines of this group.
```

```
proc MERGEOBJ(V1, CR1, V2, CR2) ;

$ This procedure equivalences the bases of composite objects
$ which are arguments of the same instruction.  However,
$ no equivalencing is performed if V1 and V2 are of
$ different gross types.  CR1 and CR2 must be the pseudo
$ creation points of V1 and V2, respectively.

$ In order to take interprocedural calls into account, this
$ routine calls the routine PARTITION to partition CR1 and
$ CR2 according to the points from which the routine
$ containing V1 and V2 is called.  The values PCR1 and
$ PCR2 returned by the routine PARTITION are the mappings
$ which map the points, from which the routine containing
$ V1 and V2 is called, to the pseudo creation points in CR1
$ and CR2.  Elements in the image sets of PCR1 and PCR2
$ are then equivalenced.

$ This routine is called by the routine GENLOC and calls the
$ routine MERGE and the routine PARTITION.

$ The global variables referenced by this routine include
$      IS-GLOB(V)  - indicates whether V is a global variable
$      IS-FORMAL(B) - indicates whether B is a formal base

$ The macros used in this routine include
$      MGTYP(OI)   - gross type of occurrence OI, see (M26).
$      OI-NAME(OI) - name of occurrence OI, see (M8).
$      ELMBASE(OI) - base of the elements of occurrence OI,
$                  - see (M28).
```

```
$ The local variables defined in this routine are

 repr
      V1,V2,OBJ,OI : ∈OI-BASE ;  $ variable occurrences
      CR1,CR2 : set([∈RC-BASE,∈OI-BASE]) ;
                                 $ pseudo creation points
      VS1,VS2 : set(∈OI-BASE) ;  $ set of variable occurrences
      CALL : ∈RC-BASE ;          $ RC-string
      PCR1,PCR2 : mmap(∈RC-BASE)set(∈OI-BASE) ;
                                 $ maps from RC-strings to sets
                                 $ of OI ; generated by the
                                 $ routine PARTITION
      CL : set(∈OI-BASE) ;       $ set of variable occurrences
 end repr ;

if MGTYP(V1) /= MGTYP(V2) then
    $ Return if V1 and V2 are of different gross types.
    return ;
end if ;

if IS-GLOB(OI-NAME(V1)) or IS-GLOB(OI-NAME(V2)) then

    $ If either V1 or V2 is a global variable then turn off
    $ IS-FORMAL flags for the bases of V1 and V2.

    IS-FORMAL(ELMBASE(V1)) := FALSE ;
    IS-FORMAL(ELMBASE(V2)) := FALSE ;

    $ Merge the bases of V1, V2 and the occurrences in CR1
    $ and CR2.
```

```
        VS1 := {OI, [-,OI]∈CR1} ;
        VS2 := {OI, [-,OI]∈CR2} ;
        MERGE((VS1+VS2) with V2, V1) ;

    else

        $ Otherwise, V1 and V2 are argument variables or
        $ variables local to a procedure.

        $ Partition CR1 and CR2 into equivalence classes,
        $ according to the points-of-call through which they
        $ transmit their values to V1 and V2.  In the case of
        $ very local variables, only one partition is
        $ produced, because all RC-strings in PS-CRTHIS are
        $ empty (values are generated within the procedure
        $ itself) .

        PCR1 := PARTITION(CR1) ;
        PCR2 := PARTITION(CR2) ;

        $ Check to see if both variables are very local.  If so,
        $ their bases are not formal.  Note that V1 (or V2) is
        $ very local if and only if PCR1 (or PCR2) is only
        $ defined on the NULL-PATH.

        if DOMAIN PCR1={NULL-PATH} and DOMAIN PCR2={NULL-PATH} then
            IS-FORMAL(ELMBASE(V1)) := FALSE ;
            IS-FORMAL(ELMBASE(V2)) := FALSE ;
        end if ;

        $ Now merge the bases appearing in each class of pseudo
```

```
    $ creation points.

    (∀ CL := PCR1{CALL} )
                              $ CALL is a point-of-call.

        if CALL = NULL-PATH then

            $ For pseudo creation points in the routine
            $ containing V1 and V2, merge the bases
            $ appearing in the pseudo creation points and
            $ the bases of V1 and V2.

            MERGE((CL+PCR2{NULL-PATH}) with V2, V1) ;

        else

            $ For pseudo creation points which are in the
            $ different routine from V1, merge the bases
            $ appearing in pseudo creation points
            $ according to their points-of-call.

            $ Choose an element having the same gross type
            $ as V1 as the representative of its class.

            if ∃ OBJ∈CL | MGTYP(OI)=MGTYP(V1) then
                MERGE(CL+PCR2{CALL}, OBJ) ;
            end if ;

    end ∀ CL ;
end if IS-GLOB;

return ;
```

```
end proc MERGEOBJ ;


proc PROPELMT(V1, CR1, V2, CR2) ;

$ This procedure handles set insertion and membership
$ operations.  V1 is a composite object, and V2 must be an
$ element of its base.  We generate locate instructions
$ for elements of CR2, and merge the elements of CR1 as in
$ the previous procedure.

$ This routine is called by the routine GENLOC and calls the
$ routines MERGE, PARTITION and INSERTLOCS.

$ The global variables referenced by this routine include
$      IS¬GLOB(V)  - indicates whether V is a global variable
$      IS¬FORMAL(B) - indicates whether B is a formal base
$      MODE(OI)    - mode of occurrence OI

$ The macros used in this routine include
$      MGTYP(OI)   - gross type of occurrence OI, see (M26).
$      OI¬NAME(OI) - name of occurrence OI, see (M8).
$      COMPTYP(M)  - element mode of mode descriptor M, see (M19).
$      ELMBASE(OI) - base of the elements of occurrence OI,
$                  - see (M28).

$ The local variables defined in this routine are

 repr
      V1,V2,OBJ,OI : ϵOI¬BASE ;  $ variable occurrences
      CR1,CR2 : set([ϵRC¬BASE,ϵOI¬BASE]) ;
```

```
                                    $ pseudo creation points
      VS1,VS2 : set(∈OI¬BASE) ;   $ set of variable occurrences
      CALL : ∈RC¬BASE ;           $ RC-string
      PCR1,PCR2 : mmap(∈RC¬BASE)set(∈OI¬BASE) ;
                                    $ maps from RC-strings to sets
                                    $ of OI ; generated by the
                                    $ routine PARTITION
      CL : set(∈OI¬BASE) ;        $ set of variable occurrences
 end repr ;

$ Assign V2 the element mode of V1.

MODE(V2) := COMPTYP(MODE(V1)) ;

if IS¬GLOB(OI¬NAME(V1)) or IS¬GLOB(OI¬NAME(V2)) then

    $ If either V1 or V2 is a global variable then turn
    $ off IS¬FORMAL flags for the bases of V1.

    IS¬FORMAL(ELMBASE(V1)) := FALSE ;

    $ Generate locate instructions to insert the elements of
    $ CR2 into the base of V1.

    VS2 := {OI, [-,OI]∈CR2} ;
    INSERTLOCS(VS2, ELMBASE(V1)) ;

    $ Merge the base of V1 and the bases of the occurrences
    $ in CR1.

    VS1 := {OI, [-,OI]∈CR1} ;
```

```
    MERGE(VS1, V1) ;

else

    $ Partition CR1 and CR2 into equivalence classes,
    $ according to points-of-call.

    PCR1 := PARTITION(CR1) ;
    PCR2 := PARTITION(CR2) ;

    $ Check to see if both variables are very local.  If so,
    $ their bases are not formal.  Note that V1 (or V2) is
    $ very local if and only if PCR1 (or PCR2) is only
    $ defined on the NULL-PATH.

    if DOMAIN PCR1={NULL-PATH} and DOMAIN PCR2={NULL-PATH} then
        IS-FORMAL(ELMBASE(V1)) := FALSE ;
        IS-FORMAL(ELMBASE(V2)) := FALSE ;
    end if ;

    $ Merge the bases and generate locate instructions
    $ according to the relevant points-of-call.

    (∀CL := PCR1{CALL} )
        if CALL=NULL-PATH then

            $ For pseudo creation points of V2 which are in
            $ the same routine as V1, generate locate
            $ instructions to insert the created values
            $ into the base of the elements of V1.

            INSERTLOCS(PCR2{CALL},ELMBASE(V1)) ;
```

```
                $ Merge the domain bases of pseudo creation
                $ points of V1 and the domain base of V1.

                MERGE(CL,V1) ;

            else

                $ For pseudo creation points which are in the
                $ different routine from V1, choose an element
                $ of the same gross type as V1 as the
                $ representative of the class, and perform
                $ locate generation and base merging.

                if ∃ OBJ ∈ CL | MGTYP(OBJ)=MGTYP(V1) then
                   INSERTLOCS(PCR2(CALL), ELMBASE(OBJ))
                   MERGE(CL, OBJ) ;
                end if ;
            end if CALL ;

        end ∀CL ;
    end if IS-GLOB;

    return ;
    end proc PROPELMT ;


    proc PROPOFMAP(V1, CR1, V2, CR2) ;

    $ This procedure processes a map retrieval operation :

    $       Y := F(X) ;
```

```
$ V1 is the map F, and V2 the logical index X.  We generate
$ 'locate' instructions to insert the occurrences
$ appearing in CR2 into the doamin base of V1, and merge
$ the bases of all occurrences appearing in CR1.  The code
$ for this procedure is identical to that for PROPELMT,
$ except for the use of the domain base of V1, instead of
$ the element base which appears in the set case.

$ This routine is called by the routine GENLOC and calls the
$ routines MERGE, PARTITION and INSERTLOCS.

$ The global variables referenced by this routine include
$      MODE(OI)     - mode of occurrence OI
$      IS¬GLOB(V)   - indicates whether V is a global variable
$      IS¬FORMAL(B) - indicates whether B is a formal base

$ The macros used in this routine include
$      MGTYP(OI)    - gross type of occurrence OI, see (M26).
$      OI¬NAME(OI)  - name of occurrence OI, see (M8).
$      DOMTYP(M)    - domain mode of mode descriptor M, see (M22).
$      DOMBASE(OI)  - domain base of occurrence OI, see (M29).
$      RANBASE(OI)  - range base of occurrence OI, see (M30).

$ The local variables defined in this routine are

 repr
      V1,V2,OBJ,OI : ϵOI¬BASE ;  $ variable occurrences
      CR1,CR2 : set([ϵRC¬BASE,ϵOI¬BASE]) ;
                                 $ pseudo creation points
      VS1,VS2 : set(ϵOI¬BASE) ;  $ set of variable occurrences
```

```
        CALL : ∈RC-BASE ;           $ RC-string
        PCR1,PCR2 : mmap(∈RC-BASE)set(∈OI-BASE) ;
                                    $ maps from RC-strings to sets
                                    $ of OI ; generated by the
                                    $ routine PARTITION
        CL : set(∈OI-BASE) ;        $ set of variable occurrences
end repr ;

$ Assign V2 the element mode of the domain of V1.

MODE(V2) := DOMTYP(MODE(V1)) ;

if IS-GLOB(OI-NAME(V1)) or IS-GLOB(OI-NAME(V2)) then

    $ If either V1 or V2 is a global variable then turn
    $ off IS-FORMAL flags for the domain base and the range
    $ base of V1.

    IS-FORMAL(DOMBASE(V1)) := FALSE ;
    IS-FORMAL(RANBASE(V1)) := FALSE ;

    $ Generate locate instructions to insert the elements of
    $ CR2 into the base of V1.

    VS2 := {OI, [-,OI]∈CR2} ;
    INSERTLOCS(VS2, DOMBASE(V1)) ;

    $ Merge the base of V1 and the bases of the occurrences
    $ in CR1.
```

```
    VS1 := {OI, [-,OI]∈CR1} ;
    MERGE(VS1, V1) ;

else

    $ Otherwise, V1 and V2 are argument variables or
    $ variables local to a procedure.

    $ Partition CR1 and CR2 into equivalence classes,
    $ according to points-of-call.

    PCR1 := PARTITION(CR1) ;
    PCR2 := PARTITION(CR2) ;

    $ Check to see if both variables are very local.  If so,
    $ their bases are not formal.  Note that V1 (or V2) is
    $ very local if and only if PCR1 (or PCR2) is only
    $ defined on the NULL-PATH.

    if DOMAIN PCR1={NULL-PATH} and DOMAIN PCR2={NULL-PATH} then
        IS-FORMAL(DOMBASE(V1)) := FALSE ;
        IS-FORMAL(RANBASE(V1)) := FALSE ;
    end if ;

    $ Merge the bases and generate locate instructions
    $ according to points-of-call.

    (∀CL := PCR1{CALL})
        if CALL=NULL-PATH then

            $ For pseudo creation points of V2 which are in
            $ the same routine as V1, generate locate
```

```
            $ instructions to insert them into the domain
            $ base of V1.

            INSERTLOCS(PCR2{CALL},DOMBASE(V1)) ;

            $ Merge the domain bases of pseudo creation
            $ points and the domain base of V1.

            MERGE(CL,V1) ;

        else

            $ For pseudo creation points which are in the
            $ different routine from V1, choose an element
            $ of the same gross type as V1 as the
            $ representative of the class, and perform
            $ locate generation and base merging.

            if ∃ OBJ ∈ CL | MGTYP(OBJ)=MGTYP(V1) then
               INSERTLOCS(PCR2{CALL}, DOMBASE(OBJ)) ;
               MERGE(CL,*OBJ) ;
            end if ;
        end if ;

    end ∀CL ;
end if IS-GLOB ;

return ;
end proc PROPOFMAP ;


proc PROPSOFMAP(VO, CRO, V1, CR1, V2, CR2) ;
```

```
$ This procedure processes the instruction :

$      F(X) := Y ;

$ V0, V1 and V2 correspond to F, X and Y respectively.

$ As before, locate instructions into the base of V0 are
$ generated for all occurrences in CR1.  In addition,
$ locate intructions into the range base of V0 are
$ generated for all objects in CR2.  The bases of the
$ occurrences appearing in CR0 are also merged with the
$ base of V0.

$ This routine is called by the routine GENLOC and calls the
$ routines MERGE, PARTITION and INSERTLOCS.

$ The global variables referenced by this routine include
$      IS¬GLOB(V)  - true if V is a global variable
$      IS¬FORMAL(B) - true if B is a formal base

$ The macros used in this routine include
$      MGTYP(OI)    - gross type of occurrence OI, see (M26).
$      OI¬NAME(OI) - name of occurrence OI, see (M8).
$      COMPTYP(M)   - element mode of mode descriptor M, see (M19).
$      DOMTYP(M)    - domain mode of mode descriptor M, see (M22).
$      RANTYP(M)    - range mode of mode descriptor M, see (M23).

$ The local variables defined in this routine are

 repr
      V0,V1,V2,OBJ : ∈OI¬BASE  $ variable occurrence
```

```
      CR0,CR1,CR2 : set([∈RC-BASE,∈OI-BASE]) ;
                                  $ pseudo creation points
      VS0,VS1,VS2 : set(∈OI-BASE) ;
                                  $ set of variable occurrences
      CALL : ∈RC-BASE ;           $ RC-string
      PCR1,PCR2 : mmap(∈RC-BASE)set(∈OI-BASE) ;
                                  $ maps from RC-strings to sets
                                  $ of OI ; generated by the
                                  $ routine PARTITION
      CL : set(∈OI-BASE) ;        $ set of variable occurrences
 end repr ;

$ Assign X and Y the element mode of the domain and the
$ element mode of the range of V0, respectively.

MODE(V1) := DOMTYP(MODE(V0)) ;
MODE(V2) := RANTYP(MODE(V0)) ;

if IS-GLOB(OI-NAME(V0)) or IS-GLOB(OI-NAME(V1)) then

    $ If either V0 or V1 is a global variable then turn
    $ off IS-FORMAL flags of the domain base and the range
    $ base of V0.

    IS-FORMAL(DOMBASE(V0)) := FALSE ;
    IS-FORMAL(RANBASE(V0)) := FALSE ;

    $ Generate locate instructions to insert the occurrences
    $ in CR1 and CR2 into the bases of the domain and the
    $ base of the range of V0, respectively.
```

```
    VS1 := {OI, [-,OI]∈CR1} ;
    INSERTLOCS(VS1, DOMBASE(VO)) ;
    VS2 := {OI, [-,OI]∈CR2} ;
    INSERTLOCS(VS2, RANBASE(VO)) ;

    $ Merge the base of VO and the bases of the occurrences
    $ in CRO.
    VSO := {OI, [-,OI]∈CRO} ;
    MERGE(VSO, VO) ;

else

    $ Otherwise, V1 and V2 are argument variables or
    $ variables local to a procedure.

    $ Partition CRO, CR1 and CR2 into equivalence classes,
    $ according to points-of-call.

    PCRO := PARTITION(CRO) ;
    PCR1 := PARTITION(CR1) ;
    PCR2 := PARTITION(CR2) ;

    $ Check to see if both variables are very local.  If so,
    $ their bases are not formal.  Note that V1 ( or V2 )
    $ is very local if and only if PCR1 ( or PCR2) is only
    $ defined on the NULL-PATH.

    if DOMAIN PCRO={NULL-PATH} and DOMAIN PCR2={NULL-PATH} then
        IS-FORMAL(DOMBASE(VO)) := FALSE ;
        IS-FORMAL(RANBASE(VO)) := FALSE ;
    end if ;
```

```
$ Merge the bases and generate locate instructions
$ according to points-of-call.

(∀CL := PCRO{CALL})
    if CALL=NULL-PATH then

        $ Generate locate instructions for the pseudo
        $ creation points of V1 and V2 which are in
        $ the same routine as V0, to insert them into
        $ the domain base and the range base of V0.

        INSERTLOCS(PCR1{CALL},DOMBASE(V0)) ;
        INSERTLOCS(PCR2{CALL},RANBASE(V0)) ;

        $ Merge the doimain base of V0 with the domain
        $ bases of the pseudo creation points of V0
        $ which are in the same routine as V0.

        MERGE(CL,V0) ;

    else

        $ For pseudo creation points which are in the
        $ different routine from V0, choose an element
        $ of the same gross type as V0 as the
        $ representative of the class, and perform
        $ locate generation and base merging.

        if ∃ OBJ ∈ CL | MGTYP(OBJ)=MGTYP(V0) then
           INSERTLOCS(PCR1{CALL}, DOMBASE(OBJ)) ;
           INSERTLOCS(PCR2{CALL}, RANBASE(OBJ)) ;
```

```
                MERGE(CL, OBJ) ;
            end if ;
        end if ;
    end ∀CL ;
end if IS-GLOB ;

return ;
end proc PROPSOFMAP ;


proc PROPOFTUP(V1, CR1, V2) ;

$ This procedure processes a tuple retrieval operation :

$     Y := T(I) ;

$ V1 corresponds the tuple T, and V2 the index.  We merge
$ the bases of all occurrences in CR1 and V1.

$ This routine is called by the routine GENLOC.

$ This routine calls the routines MERGE, PARTITION and
$ INSERTLOCS.

$ The global variables referenced by this routine include
$     IS-GLOB(V)  - true if V is a global variable
$     IS-FORMAL(B) - true if B is a formal base

$ The macros used in this routine include
$     MGTYP(OI)   - gross type of occurrence OI, see (M26).
$     OI-NAME(OI) - name of occurrence OI, see (M8).
```

```
$      ELMBASE(OI) - base of the elements of occurrence OI,
$                  - see (M28).
$      COMBASE(OI,n)  - base of the n-th component of OI, see (M31
$      GLTYP(OI)   - length of occurrence OI, see (M27).

$ The local variables defined in this routine are

 repr
      V1,V2,OBJ : ∈OI¬BASE      $ variable occurrence
      CR1 : set([∈RC¬BASE,∈OI¬BASE]) ;
                                $ pseudo creation points
      VS1 : set(∈OI¬BASE) ;     $ set of variable occurrences
      CALL : ∈RC¬BASE ;         $ RC-string
      PCR1 : mmap(∈RC¬BASE)set(∈OI¬BASE) ;
                                $ maps from rc¬strings to sets
                                $ of OI ; generated by the
                                $ routine PARTITION
      CL : set(∈OI¬BASE) ;      $ set of variable occurrences
 end repr ;

case MGTYP(V1) of

(THTUP) :          $ V1 is a tuple of unknown length.

if IS¬GLOB(OI¬NAME(V1)) then

    $ If V1 is a global variable then turn off IS¬FORMAL
    $ flag for V1.

    IS¬FORMAL(ELMBASE(V1)) := FALSE ;
```

```
    $ Merge the base of V1 and the bases of the occurrences
    $ in CR1.

    VS1 := {OI, [-,OI]∈CR1} ;
    MERGE(VS1, V1) ;

else

    $ Partition CR1 into equivalence classes, according to
    $ points-of-call.

    PCR1 := PARTITION(CR1) ;

    $ If V1 is a local variable, its base is not formal.

    if DOMAIN PCR1 = {NULL¬PATH} then
        IS¬FORMAL(ELMBASE(V1)) := FALSE ;
    end if ;

    $ Merge bases according to points-of-call.

    (∀CL := PCR1{CALL} )'
        if CALL=NULL¬PATH then

            $ For the pseudo creation points of V1 which are
            $ in the same routine as V1, merge their bases
            $ with the base of V1.

            MERGE(CL,V1) ;

        else

            $ For pseudo creation points which are in the
```

```
            $ different routine from V1, choose an element
            $ of the same gross type as V1 as the
            $ representative of the class, and perform
            $ base merging.

            if ∃ OBJ ∈ CL | MGTYP(OBJ)=MGTYP(V1) then
               MERGE(CL, OBJ) ;
            end if ;
        end if ;
    end ∀CL ;
end if IS¬GLOB;

(TMTUP) :          $ V1 is a tuple of known length.

if IS¬GLOB(OI¬NAME(V1)) then
    (∀IX := 1...GLTYP(V1))

        $ If V1 is a global variable, the bases of its
        $ components are not formal.

        IS¬FORMAL(COMBASE(V1,IX)) := FALSE ;
    end ∀ ;

    $ Merge the bases of the components of V1 and the bases
    $ of the occurrences in CR1.

    VS1 := {OI, [-,OI]∈CR1} ;
    MERGE(VS1, V1) ;

else

    $ Partition CR1 into equivalence classes, according to
```

```
$ points-of-call.

PCR1 := PARTITION(CR1) ;

$ If V1 is a local variable, its base is not formal.

if DOMAIN PCR1 = {NULL-PATH} then
    (∀IX := 1...GLTYP(V1)) IS-FORMAL(COMBASE(V1, IX)) := NO;
end if ;

$ Merge bases according to points-of-call.

(∀CL := PCR1{CALL} )
    if CALL=NULL-PATH then

        $ For the pseudo creation points of V2 which are
        $ in the same routine as V1, merge their bases
        $ with the base of V1.

        MERGE(CL,V1) ;

    else

        $ For pseudo creation points which are in the
        $ different routine from V1, choose an element
        $ of the same gross type as V1 as the
        $ representative of the class, and perform
        $ base merging.

        if ∃ OBJ ∈ CL | MGTYP(OBJ)=MGTYP(V1) then
           MERGE(CL, OBJ) ;
        end if ;
```

```
        end if ;
    end VCL ;
end if IS-GLOB ;
end case ;
return ;
end proc PROPOFTUP ;

proc PROPSOFTUP(VO, CRO, V1, V2, CR2) ;
$ This procedure handles tuple assignments.  If a tuple is
$ homogeneous, the process used is identical to that for set
$ insertion, and the procedure PROPELMT is invoked.
$ Otherwise, the integer value of the index V1 is known,
$ and the corresponding base of VO must be used to locate
$ occurences in CR2.

$ This routine is called by the routine GENLOC.

$ This routine calls the routines MERGE, PROPELMT, PARTITION
$ and INSERTLOCS.

$ The global variables referenced by this routine include
$     MODE(OI)     - mode of occurrence OI
$     IS-GLOB(V)   - true if V is a global variable
$     IS-FORMAL(B) - true if B is a formal base

$ The macros used in this routine include
$     MGTYP(OI)    - gross type of occurrence OI, see (M26).
```

```
$      OI¬NAME(OI) - name of occurrence OI, see (M8).
$      OI¬VALUE(OI) - value of occurrence OI, see (M9).
$      CTYPN(M,I)  - I-th component mode of mode descriptor M, see
$                  - (M20).
$      COMBASE(OI,N)  - base of the N-th component of OI, see (M31
$      GLTYP(OI)   - length of occurrence OI, see (M27).
$      COMPTYP(M)  - element mode of mode descriptor M, see (M19).
$      RANTYP(M)   - range mode of mode descriptor M, see (M23).

$ The local variables defined in this routine are

 repr
      V0,V1,V2,OBJ : ∈OI¬BASE  $ variable occurrence
      CR0,CR2 : set([∈RC¬BASE,∈OI¬BASE]) ;
                               $ pseudo creation points
      VS0,VS2 : set(∈OI¬BASE) ;  $ set of variable occurrences
      CALL : ∈RC¬BASE ;        $ RC-string
      PCR0,PCR2 : mmap(∈RC¬BASE)set(∈OI¬BASE) ;
                               $ maps from RC-strings to sets
                               $ of OI ; generated by the
                               $ routine PARTITION
      CL : set(∈OI¬BASE) ;     $ set of variable occurrences
 end repr ;

$ If V0 is a tuple of unknown length, invoke the routine
$ PROPELMT and return.

if MGTYP(V0)=THTUP then PROPELMT(C0,CR0,V2,CR2) ; return ; end;
```

```
INDEX := OI~VALUE(V1) ;          $ value of V1

$ Assign V2 the mode of the V1-th component of VO.

MODE(V2) := CTYPN(MODE(VO), INDEX) ;

if IS~GLOB(OI~NAME(VO)) then
    (∀IX := 1...GLTYP(VO))

        $ If VO is a global variable, the bases of its
        $ components are not formal.

        IS~FORMAL(COMBASE(VO,IX)) := FALSE ;
    end ∀ ;

    $ Generate locate instructions to insert the occurrences
    $ in CR2 into the proper base of VO.

    VS2 := {OI, [-,OI]∈CR2} ;
    INSERTLOCS(VS2, COMBASE(VO, INDEX)) ;

    $ Merge the base of VO and the bases of the occurrences
    $ in CRO.

    VSO := {OI, [-,OI]∈CRO} ;
    MERGE(VSO, VO) ;

else

    $ Partition CRO and CR2 into equivalence classes,
    $ according to points-of-call.

    PCRO := PARTITION(CRO) ;
```

```
PCR2 := PARTITION(CR2) ;

$ Check to see if both variables are very local.  If so,
$ their bases are not formal.

if DOMAIN PCRO = {NULL-PATH} then
  (∀IX:=1...GLTYP(VO)) IS-FORMAL(COMBASE(VO, IX)) := FALSE;;
end if ;

$ Merge bases and generate locate instructions according
$ to points-of-call.

(∀CL := PCRO{CALL})
    if CALL=NULL-PATH then

        $ For the pseudo creation points of V2 which are
        $ in the same routine as VO, generate locate
        $ instructions to insert them into the base of
        $ the components of VO.

        INSERTLOCS(PCR2{CALL},COMBASE(VO, INDEX)) ;

        $ Merge the domain bases of pseudo creation
        $ points and the domain base of VO.

        MERGE(CL,VO) ;

    else

        $ For pseudo creation points which are in the
        $ different routine from VO, choose an element
        $ of the same gross type as VO as the
```

```
            $ representative of the class, and perform
            $ locate generation and base merging.

            if ∃ OBJ ∈ CL | MGTYP(OBJ)=MGTYP(V1) then
               INSERTLOCS(PCR2{CALL}, COMBASE(OBJ, INDEX)) ;
               MERGE(CL, OBJ) ;
            end if ;
         end if ;
      end ∀ CL ;

end if IS-GLOB ;

return ;
end proc PROPSOFTUP ;


proc PROPSOFAMAP(VO, CRO, V1, CR1, V2, CR2) ;

$ This procedure handles the instruction :

$      F{X} := S ;

$ Two cases arise :

$ A) S is of type 'set'   Then it is a subset of the range
$ of F, and its base must be merged with the range base of
$ F.  In this case, X is handled in the same fashion as in
$ the single-valued storage case.

$ B) S is of type 'map'.  This will be the case when F is
$ actually a function of two variables, whose mode will
```

```
$ emerge as :

$      mmap{∈B}map(∈B2)*

$ The range base of F therefore contains maps, and S is an
$ element of it (instead of being a subset, as in the
$ preceding case).  S must therefore be 'located' in the
$ range base of F.  The instruction is treated in much the
$ same way as a single-valued storage operation.

$ This routine is called by the routine GENLOC.

$ This routine calls the routines MERGE, EQUIV, PROPSOFMAP,
$ PARTITION and INSERTLOCS.

$ The global variables referenced by this routine include
$      MODE(OI)     - mode of occurrence OI
$      IS¬GLOB(V)   - true if V is a global variable
$      IS¬FORMAL(B) - true if B is a formal base

$ The macros used in this routine include
$      MGTYP(OI)    - gross type of occurrence OI, see (M26).
$      OI¬NAME(OI)  - name of occurrence OI, see (M8).
$      ELMBASE(OI)  - base of the elements of occurrence OI,
$                   - see (M28).
$      DOMBASE(OI)  - domain base of occurrence OI, see (M29).
$      RANBASE(OI)  - range base of occurrence OI, see (M30).
$      COMPTYP(M)   - element mode of mode descriptor M, see (M19).
$      DOMTYP(M)    - domain mode of mode descriptor M, see (M22).
$      RANTYP(M)    - range mode of mode descriptor M, see (M23).
```

```
$ The local variables defined in this routine are

 repr
      VO,V1,V2,OBJ : ∈OI-BASE  $ variable occurrence
      VSO,VS1,VS2 : set(∈OI-BASE) ;
                               $ set of variable occurrences
      CRO,CR1,CR2 : set([∈RC-BASE,∈OI-BASE]) ;
                               $ psseudo creation points
      CALL : ∈RC-BASE ;        $ RC-string
      PCRO,PCR1,PCR2 : mmap(∈RC-BASE)set(∈OI-BASE) ;
                               $ maps from rc-strings to sets
                               $ of OI ; generated by the
                               $ routine PARTITION
      CL : set(∈OI-BASE) ;     $ set of variable occurrences
 end repr ;

$ If V2 is a map, call the routine PROPSOFMAP and return.

if MGTYP(V2)=TMAP then
    PROPSOFMAP(CO, CRO, V1, CR1, V2, CR2) :
    return ;
end if ;

$ Otherwise, assign V1 the mode of the elements of the
$ domain of VO.

MODE(V1) := DOMTYP(MODE(VO)) ;

$ Equivalence the range base of VO and the domain base of
$ V2.
```

```
EQUIV(RANBASE(VO), ELMBASE(V2)) ;

if IS-GLOB(OI-NAME(VO)) then

    $ If VO is a global variable, its domain base and range
    $ base are not formal.

    IS-FORMAL(DOMBASE(VO)) := FALSE ;
    IS-FORMAL(RANBASE(VO)) := FALSE ;


    $ Generate locate instructions to insert the occurrences
    $ in CR1 into the domain base of VO.

    VS1 := {OI, [-,OI]∈CR1} ;
    INSERTLOCS(VS1, DOMBASE(VO)) ;

    $ Merge the base of VO and the bases of the occurrences
    $ in CRO.

    VSO := {OI, [-,OI]∈CRO} ;
    MERGE(VSO, VO) ;

    $ Merge the base of V2 and the bases of the occurrences
    $ in CR2.

    VS2 := {OI, [-,OI]∈CR2} ;
    MERGE(VS2, V2) ;

else

    $ Partition CRO, CR1 and CR2 into equivalence classes,
    $ according to points-of-call.
```

```
PCRO := PARTITION(CRO) ;

PCR1 := PARTITION(CR1) ;

PCR2 := PARTITION(CR2) ;

$ Check to see if both variables are very local.  If so,
$ their bases are not formal.

if DOMAIN PCRO={NULL¬PATH} and DOMAIN PCR2={NULL¬PATH} then
    IS¬FORMAL(DOMBASE(VO)) := FALSE ;
    IS¬FORMAL(RANBASE(VO)) := FALSE ;
end if ;

$ Merge the bases and generate locate instructions
$ according to points-of-call.

(∀CL := PCRO{CALL})
    if CALL=NULL¬PATH then
        INSERTLOCS(PCR1{CALL}, DOMBASE(VO)) ;
        MERGE(CL, VO) ;
        MERGE(PCR2{CALL}, V2) ;
    else
        INSERTLOCS(PCR1{CALL}, DOMBASE(OBJO)) ;
        if ∃OBJO | MGTYP(OBJO)=MGTYP(VO) then
            MERGE(CL, OBJO) ;
        end if ;
        if ∃OBJ2 | MGTYP(OBJ2)=MGTYP(V2) then
            MERGE(PCR2{CALL}, OBJ2) ;
        end if ;
    end if ;
```

```
    end ∀ CL;
end if IS¬GLOB ;

return ;
end proc PROPSOFAMAP ;


proc MERGE(VARSET, OBJ) ;

$ This procedure equivalences the bases of the objects in
$ VARSET, which have the same gross type as OBJ, to the
$ corresponding bases of OBJ.  OBJ is always a composite
$ object.  The bases of the objects in VARSET which have
$ different gross type from OBJ are not equivalenced with
$ the base of OBJ.  If OBJ is a set or a tuple of known
$ length, a single base from each object is involved.  If it
$ is a map, then domain and range bases are equivalenced
$ seperately.  For known length mixed tuples, one base per
$ component is involved.

$ This routine is called by the routines MERGEOBJ, PROPELMT,
$ PROPOFMAP, PROPOFTUP, PROPSOFMAP, PROPSOFTUP and
$ PROPSOFAMAP.

$ This routine calls the routine EQUIV.

$ The macros used in this routine include
$      MGTYP(OI)    - gross type of occurrence OI, see (M26).
$      ELMBASE(OI) - base of the elements of occurrence OI,
$                   - see (M28).
```

```
$      DOMBASE(OI) - domain base of occurrence OI, see (M29).
$      RANBASE(OI) - range base of occurrence OI, see (M30).
$      COMBASE(OI,N)  - base of the N-th component of OI, see (M31
$      GLTYP(OI)   - length of occurrence OI, see (M27).

$ The local variables defined in this routine are

 repr
      CRSET : set([∈RC¬BASE,∈OI¬BASE]) ;
                                   $ pseudo creation points
      VARSET : set(∈OI¬BASE) ;     $ set of variable occurrences
      OI,OBJ,IV : ∈OI¬BASE ;       $ variable occurrence
 end repr ;

$ For sets and tuples of unknown length :

if MGTYP(OBJ) ∈ {TSET, THTUP} then
    (∀OI ∈ VARSET)

        $ If OI and OBJ are of the same type, equivalence
        $ the element bases of OI and OBJ.

        if MGTYP(OI)=MGTYP(OBJ) then
            EQUIV(ELMBASE(OI), ELMBASE(OBJ)) ;

            $ If OI is an ovariable which receives a
            $ retrieved value form IV, merge the mode of
            $ OI with the element mode of the base of IV.

            if IS¬OVAR(OI) and OP¬CODE(OI) in OPS¬RETRIEVE
                and MGTYP(IV:=IFROM(OI,1)) in
```

```
                        {TSET,TMAP,THTUP,TMTUP} then
                    MERGE-INTO(OI,IV) ;
                end if IS-OVAR ;
            end if MGTYP ;
        end ∀ ;
$ For maps :
elseif MGTYP(OBJ)=TMAP then
    (∀OI ∈ VARSET)
        if MGTYP(OI)=TMAP then
            $ Equivalence the domain bases and the range
            $ bases of OI and OBJ, respectively.
            EQUIV(DOMBASE(OI), DOMBASE(OBJ)) ;
            EQUIV(RANBASE(OI), RANBASE(OBJ)) ;
            $ If OI is an ovariable which receives a
            $ retrieved value form IV, merge the mode of
            $ OI with the element mode of the base of IV.
            if IS-OVAR(OI) and OP-CODE(OI) in OPS-RETRIEVE
                and MGTYP(IV:=IFROM(OI,1)) in
                        {TSET,TMAP,THTUP,TMTUP} then
                MERGE-INTO(OI,IV) ;
            end if IS-OVAR ;
        end if  MGTYP ;
    end ∀ ;
```

```
$ For tuples of known length :

elseif MGTYP(OBJ) = TMTUP then

    (∀OI ∈ VARSET)
        if MGTYP(OI)=TMTUP then

            $ Equivalence the bases of each corresponding
            $ component of OI and OBJ.

            (∀ IX := 1...GLTYP(OBJ))
                EQUIV(COMBASE(OI, IX), COMBASE(OBJ, IX)) ;
            end ∀ ;

            $ If OI is an ovariable which receives a
            $ retrieved value from IV, merge the mode of
            $ OI with the element mode of the base of IV.

            if IS¬OVAR(OI) and OP¬CODE(OI) in OPS¬RETRIEVE
                and MGTYP(IV:=IFROM(OI,1)) in
                        {TSET,TMAP,THTUP,TMTUP} then
               MERGE¬INTO(OI,IV) ,
            end if IS¬OVAR ;

        end if MGTYP ;
    end ∀ ;

end if MGTYP ;

end proc MERGE ;
```

```
proc MERGE-INTO(OV,IV) ;

$ This routine merges the mode of OV with the element mode
$ of the base of IV.  This routine is called when an
$ occurrence OV receiving a retrieved value from IV is
$ determined to be domain based and hence the elements of
$ the base of IV must be given the same mode as OV.

$ This routine is called by the routines MERGE and calls the
$ routine MODEDIS.

$ The global variables referenced by this routine include
$      BMODE(B)     - mode of base B
$      MODE(OI)     - mode of occurrence OI

$ The macros used in this routine include
$      MGTYP(OI)    - gross type of occurrence OI, see (M26).
$      COMBASE(OI,N)  - base of the N-th component of OI, see (M3
$      GLTYP(OI)    - length of occurrence OI, see (M27).
$      COMPTYP(M)   - element mode of mode descriptor M, see (M19)
$      OI-OP(OI)    - opcode of the instruction containing OI,
$                   - see (M7).

$ The local variables defined in this routine are

 repr
      OV,IV : ∈OI-BASE ;            $ variable occurrence
 end repr ;

case OI-OP(OV) of
```

```
(Q1-ARB, Q1-FROM, Q1-NEXT, Q1-INEXT) :

    $ If this is an extraction from or iteration over a set,
    $ merge the mode of OV with the element mode of the
    $ base of IV.

    if MGTYP(IV) = TSET then
        COMPTYP(BMODE(ELMBASE(IV))) MODEDIS. MODE(OV) ;
    end if ;

(Q1-NEXTD, Q1-INEXTD) :

    $ If this is an iteration over a map, merge the mode of
    $ OV with the element mode of the range base of IV.

    if MGTYP(IV) = TMAP then
        COMPTYP(BMODE(RANBASE(IV))) MODEDIS. MODE(OV) ;
    end if ;

(Q1-OF) :

    case MGTYP(IV) of

    (THTUP) :

        $ If this is a value retrieval from a tuple of
        $ unknown length, merge the mode of OV with the
        $ element mode of the base of IV.

        COMPTYP(BMODE(ELMBASE(IV))) MODEDIS. MODE(OV) ;

    (TMAP) :
```

```
            $ If this is a value retrieval from a map, merge the
            $ mode of OV with the element mode of the range
            $ base of IV.

            COMPTYP(BMODE(RANBASE(IV))) MODEDIS. MODE(OV) ;

        (TMTUP) :

            $ If this is a value retrieval from a tuple of known
            $ length, merge the mode of OV with the element
            $ mode of the proper component base of IV.

            IX := OI-VALUE(IFROMO(OV,2)) ;
            COMPTYP(BMODE(COMBASE(IV,IX))) MODEDIS. MODE(OV) ;

        end case MGTYP ;

    (Q1-OFB) :

        $ If this is a range retrival from a map, merge the mode
        $ of OV with the range mode of IV.

        if MGTYP(IV) = TMAP then
            COMPTYP(BMODE(RANBASE(IV))) MODEDIS. COMPTYP(MODE(OV)) ;
        end if ;
    end case OI-OP ;

    return ;
    end proc MERGE-INTO ;


    proc EQUIV(B1, B2) ;
```

```
$ This routine equivalences the bases B1 and B2 by using the
$ compressed balanced tree technique.  Equivalence classes
$ are represented by a forest of trees.  The root of a
$ tree is the representative (called the real base of the
$ tree) of all the bases in the tree.  Trees are
$ structured by the map PARENT ; PARENT(B) points to the
$ preceding node of B in the tree if B is not a root,
$ otherwise PARENT(B) is undefined.

$ When we equivalence two bases B1 and B2, we first find the
$ roots R1 and R2 of the trees containing B1 and B2
$ respectively.  If R1 and R2 are the same (i.e., B1 and
$ B2 are in the same tree), nothing has to be done.
$ Otherwise, we link that one of R1 or R2, whose tree has
$ fewer nodes, to the other as a subtree.

$ Four global maps are defined on the roots of trees to
$ describe the properties of the corresponding real bases.

$     IS¬FORMAL(B) - indicates whether B is a formal base
$     NBASES(B)    - number of bases in the same class of B
$     NBASEDON(B) - number of the sets and maps based on B
$     BMODE(B)     - mode of base B

$ PARENT is another global map defined on bases to point to
$ the preceding nodes in the tree.

$ This routine is called by the routine MERGE and
$ INSERTLOCS.  It calls the routine REALB and MODEDIS.
```

```
$ The local variables defined in this routine are

REPR
      B1,B2,RB1,RB2 : ∈SB~BASES ;    $ bases
end repr ;

$ If the roots R1 and R2 of the trees containing B1 and B2
$ are the same, then return.

if (RB1:=REALB(B1)) = (RB2:=REALB(B2)) then
    return ;

else

    if NBASES(RB1) < NBASES(RB2) then

        $ If the tree containing RB1 has fewer nodes than
        $ the tree containing RB2 then link RB1 as a
        $ substree of RB2.

        PARENT(RB1) := RB2 ;

        $ Update NBASES and NBASEDON of the new tree.

        NBASES(RB2) + NBASES(RB1) ;
        NBASEDON(RB2) + NBASEDON(RB1) ;

        $ The new tree is a formal one if the original trees
        $ of RB1 and RB2 both are formal.

        IS~FORMAL(RB2) := IS~FORMAL(RB2) and IS~FORMAL(RB1) ;

        $ The mode of the new tree is the disjunction of the
```

```
        $ modes of the original trees.

        BMODE(RB2) := BMODE(RB2) MODEDIS. BMODE(RB1) ;

    else

        $ If the tree containing RB2 has fewer nodes than
        $ the tree containing RB1 then link RB2 as a
        $ substree of RB1.

        PARENT(RB2) := RB1 ;

        $ Update NBASES and NBASEDON of the new tree.

        NBASES(RB1) + NBASES(RB2) ;
        NBASEDON(RB1) + NBASEDON(RB2) ;

        $ The new tree is a formal one if the original trees
        $ of RB2 and RB1 both are formal.

        IS-FORMAL(RB1) := IS-FORMAL(RB1) and IS-FORMAL(RB2) ;

        $ The mode of the new tree is the disjunction of the
        $ modes of the original trees.

        BMODE(RB1) := BMODE(RB1) MODEDIS. BMODE(RB2) ;

    end if NBASES ;
end if (RB1 ;

return ;
end proc EQUIV ;
```

```
proc REALB(B) ;

$ This routine finds the real base of B.  This is achieved
$ by following the map PARENT until we reach a node of
$ which PARENT is undefined.  As a side effect, the tree
$ is compressed.  All the nodes R1 on the path from B to
$ the root R2, except the node which is an immediate
$ descendant of the root, are re-linked to the root as
$ immediate descendants, i.e., PARENT(R1) is set to point to
$ R2.  This compression procedure reduces the depth of the
$ tree and makes subsequent root finding procedures more
$ efficient.

$ This routine is called by the routine EQUIV.

$ The only global variable referenced by this routine is
$      PARENT(B)    - preceding node of B in the tree

$ The local variables defined in this routine are

 repr
      B,R1,R2,R3 : ∈SB-BASE ;      $ bases
      WORK : set(∈SB-BASE) ;       $ workpile of bases
end repr ;

$ Initialize workpile.
WORK := nl ;

$ Let R1 point to B.
R1 := B ;
```

```
$ If B is a root, then return.  Otherwise, let R2 point to
$ the preceding node of R1.

if (R2:=PARENT(R1)) = OM then
    return ;
end if ;

$ While R2 is not a root, insert R1 into WORK and let R2
$ point to the preceding node of R1.

(while R3:=PARENT(R2) /= OM)
    WORK with R1 ;
    R1 := R2 ;
    R2 := R3 ;
end while ;

$ Compress the tree by re-linking the nodes in WORK to R2.

(∀ R1 ∈ WORK)
    PARENT(R1) := R2 ;
end ∀ ;

return R2 ;
end proc REALB ;


proc INSERTLOCS(VARSET, BASE) ;

$ This procedure ensures the variable occurrences appearing
$ in VARSET to be elements of the base BASE by either
$ merging the bases of the occurrences appearing in VARSET
$ with BASE or generating locate instructions to insert
```

```
$ the occurrences in VARSET into BASE.  If an occurrence X
$ in VARSET is the ovariable of a value retrieval
$ instruction of which the composite object, from which the
$ value is retrieved, has been domain based on a base B,
$ then B is equivalenced with BASE.  If an occurrence X in
$ VARSET has already received a 'locate' instruction to
$ insert its value into a base B, then B is equivalenced
$ with BASE in the same way as in the previous case.  For
$ other occurrences in VARSET, locate instructions are
$ generated to insert their values into BASE.  Generated
$ locate instrcuctions are collected into the global map
$ LOCINS.  LOCINS maps each occurrence to the bases into
$ which the occurrence is to be inserted.

$ This routine is called by the routines MERGEOBJ, PROPELMT,
$ PROPOFMAP, PROPOFTUP, PROPSOFMAP, PROPSOFTUP and
$ PROPSOFAMAP.

$ This routine calls the routine EQUIV.

$ The only global variable referenced by this routine is
$     LOCINS(OI)  - the base into which OI is inserted

$ The local variables used in this routine are

 repr
      VARSET : set(∈OI¬BASE) ;  $ set of variable occurrences
      OI,OBJ : ∈OI¬BASE ;       $ variable occurrence
 end repr ;
```

```
(∀OI ∈ VARSET)

    if IS-OVAR(OI) and OP-CODE(OI) in OPS-RETRIEVE
        and MGTYP(IV:=IFROM(OI,1)) in {TSET,TMAP,THTUP,TMTUP} the

        $ If OI is the ovariable of a value retrieval
        $ instruction, equivalence BASE with the base of
        $ the ivariable.

        case OI-OP(OI) of

        (Q1-ARB, Q1-FROM, Q1-NEXT, Q1-INEST) :

            if MGTYP(IV) = TSET then
                EQUIV(BASE,ELMBASE(IV)) ;
            end if ;

        (Q1-NEXTD, Q1-INEXTD) :

            if MGTYP(IV) = TMAP then
                EQUIV(BASE,DOMBASE(IV)) ;
            end if ;

        (Q1-OF) :

            case MGTYP(IV) of

            (THTUP) :
                EQUIV(BASE,ELMBASE(IV)) ;

            (TMAP) :
                EQUIV(BASE,RANBASE(IV)) ;
```

```
            (TMTUP) :
                IX := OI->VALUE(IFROMO(OI,2)) ;
                EQUIV(BASE,COMBASE(IV,IX)) ;
            end case MGTYP ;
        end case OI->OP ;
        $ Update the element mode of BASE to indicate that
        $ OI is an element of BASE.
        COMPTYP(BMODE(BASE)) MODEDIS. MODE(OI) ;
    elseif LOCINS(OI) /= OM then
        $ If OI has been inserted into a base, then
        $ equivalence this base with BASE.
        EQUIV(BASE, LOCINS(OI)) ;
    else
        $ Otherwise, insert OI into BASE.
        LOCINS(OI) := BASE ;
        $ Update the element mode of BASE to indicate that
        $ OI is an element of BASE.
        COMPTYP(BMODE(BASE)) MODEDIS. MODE(OI) ;
    end if ;
end V ;
return ;
```

```
end proc INSERTLOCS ;


proc M1 MODEDIS. M2 ;

$ This procedure evaluates the disjunction of two mode
$ descriptors.  It differs from the disjunction routine in
$ the SETL typefinder, in that it handles element-of-base
$ descriptors.

$ The rules for disjunction of modes are as follows :

$ A) The disjunction of different gross types yields
$ 'general'.

$ B) The disjunction of two element-of-base modes yields an
$ element mode, and has the side-effect of equivalencing
$ the two bases.

$ C) The disjunction of two sets yields a set whose elements
$ are the disjunction of the respective element
$ descriptors of the two sets.

$ D) The disjunction of two maps yields a map whose domain
$ and range modes are the disjunction of the domain modes
$ and the range modes of the two maps.

$ E) The disjunction of two tuples yields a tuple whose
$ component modes are the disjunction of the corresponding
$ component modes of the two tuples.
```

```
$ This recursive routine is called by the routine
$ MERGE-INTO, INSERTLOCS and EQUIV.  It calls the routine
$ EQUIV.

$ The macros used in this routine include
$      GROSSTYP(M) - gross type of mode M, see (M19).
$      IS-PRIM(M)  - true if M is a primitive mode, see (M17).
$      BASENAM(M)  - base name of mode descritor M, see (M24).
$      COMPTYP(M)  - element mode of mode descriptor M, see (M19)
$      DOMTYP(M)   - domain mode of mode descriptor M, see (M22).
$      RANTYP(M)   - range mode of mode descriptor M, see (M23).
$      CTYPN(M,I)  - I-th component mode of mode descriptor M,
$                  - see (M20).
$      LENTYP(M)   - length of mode descrptor M, see (M21).

$ The local variables defined in this routine are

 repr
      M1, M2, DISM : ∈MODE-BASE ;    $ mode desciptors
      B1, B2 : ∈SB-BASE ;            $ bases
 end repr ;


$ If M1 is of zero type, the disjunction is M2.

if GROSSTYP(M1) = TZ then
    return M2 ;

$ If M2 is of zero type, the disjunction is M1.

elseif GROSSTYP(M2) = TZ then
```

```
    return M1 ;

$ If M1 and M2 are of different gross types, their
$ disjunction is a 'general' type.

elseif GROSSTYP(M1) /= GROSSTYP(M2) then return TGEN ;

$ If M1 and M2 are of the same primitive modes, then their
$ disjunction is M1.

elseif IS~PRIM(GROSSTYP(M1)) then return M1 ;

elseif GROSSTYP(M1)=TELMT then

    $ If M1 and M2 are member basing modes, equivalence
    $ their bases and return M1.

        EQUIV(BASENAM(M1), BASENAM(M2)) ;
    return M1 ;

else

    $ Otherwise, M1 and M2 are composite modes.  Construct a
    $ template for the disjunction mode descriptor.

    DISM := [GROSSTYP(M1)] ;

    $ Construct the element mode of DISM by recursively
    $ invoking this routine to derive the disjunction of
    $ the element modes of M1 and M2.

    case GROSSTYP(M1) of

    (TSET, THTUP) :
```

```
        COMPTYP(DISM) := COMPTYP(M1) MODEDIS. COMPTYP(M2) ;

    (TMAP) :

        COMPTYP(DISM) := [TMTUP, [ ]] ;
        DOMTYP(DISM) := DOMTYP(M1) MODEDIS. DOMTYP(M2) ;
        RANTYP(DISM) := RANTYP(M1) MODEDIS. RANTYP(M2) ;

    (TMTUP) :

        (∀IX := 1...LENTYP(M1))
            CTYPN(DISM,IX):=CTYPN(M1,IX) MODEDIS. CTYPN(M2,IX)
        end ∀IX;
    end case;

    return DISM ;

end if ;

end proc MODEDIS. ;


proc PARTITION(CRSET) ;

$ This procedure takes an attribute set and partitions it
$ according to points-of-call through which attributes
$ were propagated.  The RC-string which accompanies each
$ member of the set is scanned backwards, skipping over
$ completed calls until a call without a return is found.
$ These call instructions serve to partition the attribute
$ set.  The procedure returns a map on call instructions.
```

```
$ This routine is called by the routines MERGEOBJ,
$ PROPOFMAP, PROPOFTUP, PROPSOFMAP, PROPSOFTUP and
$ PROPSOFAMAP.  It calls the routine LASTCALL.

$ The local variables defined in this routine are

 repr
      CLASSES : mmap(∈RC-BASE)set(∈OI-BASE) ;
                                   $ map RC-string into set of
                                   $ occurrences
      CRSET : set([∈RC-BASE,∈OI-BASE]);
                                   $ pseudo creation points
      P : ∈RC-BASE ;               $ RC-string
      OI : ∈OI-BASE ;              $ variable occurrence
 end repr ;

CLASSES := nl ;
(∀[P, OI] ∈ CRSET) CLASSES{LASTCALL(P)} with OI ;;
return CLASSES ;
end proc PARTITION ;


proc LASTCALL(RC-STRING) ;

$ This procedure scans RC-STRING backwards until it
$ encounters a call not matched by a return.  If RC-STRING
$ is a NULL-PATH, then it returns NULL-PATH, otherwise it
$ returns the RC-CALL found.  However, if RC-STRING is
$ incomplete or no such call is found, it also returns
```

```
$ NULL-PATH.

$ This routine is called by the routine PARTITION.

$ The local variables defined in this routine are

 repr
      RC-STRING : ∈RC-BASE ;        $ RC-string
      I : int ;                     $ length of RC-string
      CALLCOUNT : int ;             $ count
 end repr ;

$ If RC-STRING is a NULL-PATH, returns NULL-PATH.

if RC-STRING =NULL-PATH then return NULL-PATH ;

else
     I = # RC-STRING ;
     CALLCOUNT = 0 ;

     $ Scan RC-STRING backwards.

     (while CALLCOUNT<=0 and I>1 DOING I := I-1 ;)
          if RC-STRING(I)(1)=RC-CALL then

               $ If the i-th component is a call, then
               $ increment count by one.

               CALLCOUNT := CALLCOUNT +1 ;
          else

               $ Otherwise, it is a return and therefore count
```

```
            $ is decremented by one.

             CALLCOUNT := CALLCOUNT - 1 ;
        end if ;
    end while ;

    if CALLCOUNT <= 0 and I=1 then

        $ If RC-STRING is incomplete or no such call is
        $ found, return NULL-PATH.

        return NULL-PATH ;

    else

        $ Otherwise, return the call found.

        return RC-STRING(I) ;

    end if CALLCOUNT ;
end if RC-STRING ;
end proc LASTCALL ;


proc MOVELOCS ;

$ This procedure moves a 'locate' instruction out of a loop
$ whenever the basing pointer which it generates is not
$ actually used within the loop.  The following case is
$ typical : a variable X is known to be 'ϵB' ;
$ PS-CRTHIS{X} includes the following occurrence of X :

$     (∀I := 1..100) X := X + Y ;;
```

```
$ The procedure GENLOCS will have provisionally inserted a
$ locate instruction within the loop, for the ovariable X
$ therein.  This is clearly inappropriate, because no such
$ value of X, (except the last one is used as a base
$ element).  The proper place for the locate instruction
$ is at exit from the loop or from some containing loop.
$ The procedure shown below systematizes the process of
$ 'locate motion'.  A 'locate' instruction can be moved
$ out of an interval if no use is made of the basing
$ pointer which it generates, within the interval.  This
$ can be ascertained by following the FFROM map of the
$ (provisionally) located variable.  If we reach an
$ operation which uses the basing pointer within the
$ interval then the 'locate' cannot be moved.  If the use
$ appears in some successor interval, then it will be
$ advantageous to move the 'locate' operation to the head
$ of that interval.

$ The following procedure systemizes the process of
$ locate instruction motion.  We scan the FFROM chain for
$ each occurrence OI at which a locate instruction has been
$ suggested in phase II.  The scanning procedure continues
$ until we find all the places at which the basing pointer
$ created at OI might potentially be used.  The intervals
$ which contain these points are called the target intervals
$ of OI, and a map MOVETO summarizing this information is
$ generated.  If one of the target intervals of OI is the
$ interval in which OI resides, MOVETO(OI) is defined as nl.
```

```
$ We use MOVETO to insert actual locate instructions as
$ follows.  If MOVETO(OI) is nl then a locate operation is
$ inserted right after OI is created.  Otherwise, for each
$ interval INT in MOVETO(OI), a locate operation is inserted
$ at the entry to the largest interval which includes INT
$ but not OI.

$ This routine is called by the main routine AUTO-DATA.

$ This routine calls the routines INTMAX, INS-AFTER and
$ INS-TARG.

$ The global variables referenced by this routine include
$      LOCINS(OI)   - the base into which OI is inserted
$      MODE(OI)     - mode of occurrence OI
$      BASE-ELMTS   - sets of occurrences which have been
$                   - known to be elements of bases

$ The macros used in this routine include
$      IS-OVAR(OI) - true if occurrence OI is an ovariable,
$                  - see (M11).
$      IS-HASHED(OI)  - true if OI is subject to operations
$                  - involving hashing, see (M13).
$      OFROMI(OI)  - the ovariable in the same instruction
$                  - as the ivariable OI, see (M15).

$ The local variables defined in this routine are

 repr
     WORK : set(∈OI-BASE) ;      $ workpile of occurrences
```

```
    OI, WOI, U, NU, NEWOVAR, NEWIVAR : ∈OI¬BASE ;
                                    $ variable occurrences
    USES : set([∈RC¬BASE,∈OI¬BASE]) ;
                                    $ workpile
    P, NP, NNP : ∈RC¬BASE ;         $ RC-strings
    I, NEWI : int ;                 $ instruction identifiers
    MOVETO : mmap(∈OI¬BASE)set([∈RC¬BASE,∈OI¬BASE]) ;
                                    $ map occurrences to target
                                    $ intervals
    BASE : ∈SB¬BASE ;               $ base
end repr ;

$ Initialize base¬elmts.
BASE¬ELMTS := nl ;

$ For all occurrences to be inserted into bases

(∀[OI, BASE] ∈ LOCINS | not CAN¬DROP(BASE) )
    WORK := {OI} ;          $ workpile
    (while WORK/=nl)
        WOI from WORK ;
        USES := FFROM{WOI} ;

        $ We follow FFROM until we pass out of the
        $ interval, or until we find a hashing use of the
        $ variable.

        (while USES /=nl)
            [P, U] from USES ;
```

```
$ If the interval containing U is different from
$ the interval containing WOI, then prepare to
$ move locate instruction from WOI toward U.

if OI-INTOV(U)/=OI-INTOV(WOI) then
    MOVETO{WOI} with [P, U] ;

$ If U is subject to an instruction involving
$ hashing operation, locate instruction can
$ not be moved.

elseif IS-HASHED(U) then
    MOVETO{WOI} := nl ;
    continue ∀ ;

$ If the occurrence OI we are tracing is
$ assigned to another, we must also trace the
$ target of the assignment, for possible use
$ of the basing pointer thus transmitted.

elseif OI-OP(U) in OPS-ASN then
    NEWUSES := FFROM{OFROMI(U)} + FFROM{U} ;

else              $ Continue chaining.
    NEWUSES := FFROM{U} ;
end if ;

$ The elements in NEWUSES, which are actually
$ chained to the original variable must be
$ processed.
```

```
            USES + { [NNP, NU] : [NP, NU] ∈ NEWUSES |
                (NNP := P CC. NP)/= ERROR¬PATH } ;
        end while USES;
    end while WORK ;
end ∀[OI ;

$ Now perform code insertion.  First process the 'locate'
$ instructions which were not moved.  The insertion to be
$ performed is indicated by an assignment statement, from
$ an occurrence having primitive mode, to an occurrence
$ having member basing mode.

(∀[OI, BASE] ∈ LOCINS | MOVETO(OI)=nl and not CAN¬DROP(BASE) )
    I := INSTNO(OI) ;

    $ Physically insert a locate instruction after
    $ instruction I.

    NEWARGS := [OI¬NAME(OI), OI¬NAME(OI)] ;
    NEWI := INS¬AFTER(I, Q1¬ASN, NEWARGS) ;
    NEWOVAR := [NEWI,1] ;
    NEWIVAR := [NEWI,2] ;

    $ Assign proper modes to the variable occurrences in
    $ this new instruction.

    MODE(NEWOVAR) := [TELMT, NULL¬PATH, BASE] ;
    MODE(NEWIVAR) := MODE(OI) ;

    $ Update the set BASE¬ELMTS.
```

```
    BASE-ELMTS with [NEWOVAR,BASE] ;

    $ Update BFROM and FFROM.

    FFROM[NEWOVAR] := FFROM[OI] ;
    FFROM[OI] := { [NULL-PATH,NEWIVAR] } ;
    BFROM[NEWIVAR] := { [NULL-PATH,OI] } ;
end V ;

$ Now process the occurrences in MOVETO.  The optimal point
$ for inserting the new instruction is the head of the
$ largest interval which contains the target occurrence,
$ and which does not contain the original occurrence
$ (whose locate has been moved).  The problem of finding
$ such an interval also arises in relation with copy
$ optimization.  Here we use several utility procedures
$ taken from that module :

$ INTMAX : finds the largest interval in a geven sequence of
$ derived intervals which does not contain a given
$ variable occurrence.

$ INS-TARG : inserts a new instruction in the target block
$ of the chosen interval, and returns nl if such an
$ instruction is already in the target block.

(VVARSET := MOVETO[OI] )
    BASE := LOCINS(OI) ;

    (V[P, V] ∈ VARSET)
```

```
    $ Calculate intervals of the derived sequence which
    $ contain V.

    INTSEQ := [ID : ID:=OI¬INTOV(V) while ID  /=  OM
              doing ID:=INTOV(ID) ] ;

    $ Find the target block of the last interval of
    $ INTSEQ.

    TARG := INTMAX(OI, P, #INTSEQ+1, INTSEQ) ;

    $ Insert a locate instruction into this target
    $ block.

    I:=INS¬TARG(INTSEQ(TARG),Q1¬ASN,[OI¬NAME(V),OI¬NAME(V)]);

    $ Once the instruction is successfully inserted, set
    $ proper mode of new occurrecnes to indicate the
    $ locate operation to to performed.

    if I /= OM then
        NEWOVAR := [I,1] ;          $ ovariable
        NEWIVAR := [I,2] ;          $ ivariable
        MODE(NEWOVAR) := [TELMT, OM, BASE] ;
        MODE(NEWIVAR) := MODE(OI) ;

        $ Indicate that the new ovariable has element
        $ basing.

        BASE¬ELMTS with [NEWOVAR,BASE] ;
    end if ;
end ∀[P ;
```

```
end ∀VARSET ;

return ;

end proc MOVELOCS ;


proc UPDMODES ;

$ This constitutes the fourth phase of the data structure
$ choice algorithm.  We adjust the modes of occurrences in
$ three steps :

$ A) For composite objects and member based objects, we
$ adjust their modes using the procdure SUBSTMD, to
$ replace references to dropped bases by the mode of their
$ elements and also replace references to non-real bases
$ by references to real bases.

$ B) At each occurrence we determine whether member basings
$ and type information (possibly involving domain basings)
$ are useful by examining the subsequent uses of the value
$ appearing at this occurrence.  Two indicators
$ NON-HASH-USE and HASH-USE are generated for this
$ purpose.

$ C) We then propagate member basing pointers from locate
$ and value retrieval instructions to other occurrences
$ which need basings.  The propagation procedure ensures
$ that proper basings are carried with the variable
$ values.
```

```
$ This routine is called by the main routine AUTO-DATA.  It
$ calls the routines MODECMPRS, USE-DETERM and
$ BASING-PROP.

MODECOMPRS() ;
USE-DETERM() ;
BASING-PROP() ;

return ;
end proc UPDMODES ;


proc MODECOMPRS ;

$ This procedure adjusts the basing mode of variable
$ occurrences, using the procedure SUBSTMD.  Basings
$ referencing non-real bases are adjusted to reference
$ real bases.  Basings referencing bases dropped are
$ replaced by references to the element modes of the
$ bases.

$ The purpose of introducing dummy bases for tuples and for
$ the range of maps was to use these bases as markers for
$ possibly complex structures which may themselves be
$ based.  These markers are replaced by the corresponding
$ structures by means of procedure SUBSTMD described
$ above.  A frequent and important case where this
$ mechanism is useful is that of multivariate maps.  The
$ operation :
```

```
$      F(X, Y) := Z ;

$ expands into the following sequence of univariate
$ retrievals and storages,

$      T := f{X} ;
$      T(Y) := Z ;
$      F{X} := T ;

$ These generate the following basings :

$      F : map{∈B1}∈B2
$      T : map(∈B3)∈B4

$ Our system will also produce locate instructions :

$      B2 with T ;
$      B3 with Y ;
$      B4 with Z ;

$ If F is only used as a bivariate map, then B2 will
$ eventually be recognized to be useless, and after
$ determining that the mode of B2 is 'map(∈B3)∈B4' , the
$ final mode for F will be :

$      F : map{∈B1}map(∈B3)∈B4 ;

$ which is the desired descriptor.

$ In general there will be fewer modes to adjust than
$ variable occurences.  It is therefore economical to map
$ modes themselves into their final forms, and then to use
```

```
$ this map to update the modes of variables.

$ This routine is called by the routine UPDMODES and calls
$ the routine SUBSTMD.

$ The global variables referenced by this routine include
$      ALL-OI        - all variable occurrences
$      MODE(OI)      - mode of occurrence OI

$ The macros used in this routine include
$      MGTYP(OI)     - gross type of occurrence OI, see (M26).
$      IS-PRIM(M)    - true if M is a primitive mode, see (M17).

$ The local variables defined in this routine are

 repr
     NEWMODE : smap(∈MDOE-BASE)∈MODE-BASE ;
                                   $ temporary map from mode
                                   $ descriptor to mode descriptor
      OI : ∈OI-BASE ;              $ variable occurrence
      M : ∈MODE-BASE ;             $ mode descriptor
 end repr ;


$ Initialize NEWMODE.

NEWMODE := NL ;

$ Update non-primitive mode descriptors.

(∀M ∈ RANGE MODE | not IS-PRIM(GROSSTYP(M)))
     NEWMODE(M) := SUBSTMD(M) ;
```

```
end ∀ ;

$ Update the non-primitive modes of variable occurrences.

(∀OI ∈ ALL-OI | not IS-PRIM(MGTYP(OI))
    MODE(OI) := NEWMODE(MODE(OI)) ;
end ∀ ;
return ;
end proc MODECOMPRS ;


proc SUBSTMD(M) ;

$ This procedure updates mode descriptors, by replacing
$ references to dropped bases by the mode descriptors for
$ the corresponding base elements.  It also replaces base
$ names by the names of their representatives, so that the
$ updated mode contains only references to real bases .

$ As a side-effect, the modes of real bases are also updated
$ when they are referenced for the first time.  For
$ example, when we update a mode '∈B1' where the real base
$ of B1 is B2 whose mode was 'base([∈B3,bool])' and where
$ B3, whose mode was 'base(int)', is dropped, this routine
$ updates the mode of B2 to be 'base([int,bool])' and then
$ replaces the mode '∈B1' by '∈B2'.  This base mode
$ updating procedure makes subsequent mode updating
$ procedures more efficient, especially when they reference
$ to the bases which have been referenced before.
```

```
$ This recursive routine is called by the routine MODECMPRS.

$ The global variables referenced by this routine include
$      REALB(B)     - representative of the equivalence class
$      CAN-DROP(B) - true if base B can be dropped

$ The macros used in this routine include
$      GROSSTYP(M) - gross type of mode M, see (M19).
$      IS-PRIM(M)  - true if M is a primitive mode, see (M17).
$      BASENAM(M)  - base name of mode descritor M, see (M24).
$      COMPTYP(M)  - element mode of mode descriptor M, see (M19).
$      DOMTYP(M)   - domain mode of mode descriptor M, see (M22).
$      CTYPN(M,I)  - I-th component mode of mode descriptor M,
$                  - see (M20).
$      RANTYP(M)   - range mode of mode descriptor M, see (M23).
$      LENTYP(M)   - length of mode descrptor M, see (M21).

$ The local variables defined in this routine are

 repr
      B : ∈SB-BASE ;              $ base
      M : ∈MODE-BASE ;            $ mode descriptor
      IX : int ;                  $ index
 end repr ;

$ If M is a primitive mode, then return M.

if IS-PRIM(GROSSTYP(M)) then return M ;

else case GROSSTYP(M) of
```

```
    $ For a member basing mode M, update the mode of the
    $ real base B of M.  If B can be dropped then
    $ re-assign M the mode of the elements of B, otherwise
    $ complete the BASENAM component of M.

    (TELMT) :

        B := REALB(BASENAM(M)) ;
        COMPTYP(BMODE(B)) := SUBSTMD(COMPTYP(BMODE(B)) ;
        if CAN-DROP(B) then
            M := COMPTYP(BMODE(B)) ;
        else
            BASENAM(M) := B ;
        end if ;

    $ For other composite modes, update the element mode of
    $ M by recursively invoking this routine.

    (TBASE, TSET, THTUP) : COMPTYP(M) := SUBSTMD(COMPTYP(M)) ;

    (TMAP) : DOMTYP(M) := SUBSTMD(DOMTYP(M)) ;
             RANTYP(M) := SUBSTMD(RANTYP(M)) ;

    (TMTUP) :
        (∀IX :=1...LENTYP(M))
            CTYPN(M, IX) := SUBSTMD(CTYPN(M, IX)) ;
        end ∀ ;

    end case ;
return M ;
end if ;
```

```
end proc SUBSTMD ;

proc USE-DETERM ;

$ This procedure determines whether member basing and/or
$ domain basing should be carried at variable occurrences.
$ If the value flow shows that the value of an occurrence
$ will subsequently be subject to membership test
$ (explicitly or implicitly), member basing pointers
$ should be kept ; this is flagged by setting HASH-USE
$ equal to true.  Similarly, if the value flow shows that
$ the type information or domain basing is useful in the
$ subsequent uses, type information (with domain basing)
$ should be kept ; this is flagged by setting NON-HASH-USE
$ true.  Note that both types of information might be
$ carried along with certain occurrences.

$ This routine is called by the routine UPDMODES.

$ The global variables referenced by this routine include
$      ALL-IVAR     - all ivariables
$      BFROM(OI)    - occurrences to which OI is directly linked
$      HASH-USE(OI) - true if member basing should be kept for
$                   - occurrence OI
$      NON-HASH-USE(OI) - true if domain basing should be kept for
$                   - occurrence OI

$ The macros used in this routine include
$      MGTYP(OI)    - gross type of occurrence OI, see (M26).
```

```
$      IS-OVAR(OI) - true if occurrence OI is an ovariable,
$                  - see (M11).
$      OI-OP(OI)   - opcode of the instruction which contains
$                  - occurrence OI, see (M7).
$      IFROMO(OV,I) - I-th ivariable of the instruction which
$                  - contains ovariable OV, see (M15).

$ The local variables defined in this routine are

 repr
      WORK : set(∈OI-BASE)          $ workpile of occurrences
      IV, OI : ∈OI-BASE ;           $ variable occurrences
      POI : [∈RC-BASE,∈OI-BASE] ;
                                    $ pair of RC-string and occurrenc
      P : ∈RC-BASE ;                $ RC-string
 end repr ;

CONST U-MEMB,U-TYPE ; end ;
                    $ usage information to be propagated

$ Initialize WORK, HASH-USE and NON-HASH-USE.

WORK := NL ;
HASH-USE := NL ;
NON-HASH-USE := NL ;

( ∀ IV ∈ ALL-IVAR )

     $ The ivariables which need member basing pointers
     $ have been assigned TELMT mode.
```

```
    if MGTYP(IV) = TELMT then
        HASH-USE(IV) := TRUE ;

        $ Prepare to propagate information backwards through
        $ the BFROM chain.

        WORK + { [POI,U-MEMB] : POI∈BFROM(IV) } ;

    $ Otherwise, IV must be subject to an operation
    $ involving global structure iteration, unless this is
    $ an assignment instruction.  In such a case, type
    $ information (possibly involving domain basing) is
    $ generally useful.

    elseif OI-OP notin OPS-ASN then
        NON-HASH-USE(IV) := TRUE ;

        $ Prepare to propagate information backwards through
        $ the BFROM chain.

        WORK + { [POI,U-TYPE] : POI∈BFROM(IV) } ;
    end if ;

end ∀ ;

$ Usage infromation is then propagated backward through
$ BFROM chain and assignment instructions.

( while WORK /= NL )

    [[P,OI],USE] from WORK ;
```

```
$ Determine usage information for the ovariables and
$ the ivariables of assignment instructions.

if IS-OVAR(OI) or OI-OP(OI) in OPS-ASN then

    case USE of

    (U-MEMB) :

        $ If HASH-USE flag of OI is already on then OI
        $ need not be processed, otherwise turn on the
        $ flag.

        if HASH-USE(OI)  then
            continue while ;
        else
            HASH-USE(OI) := TRUE ;
        end if ;

    (U-TYPE) :

        $ If NON-HASH-USE flag of OI is already on then
        $ OI need not be processed, otherwise turn on
        $ the flag.

        if NON-HASH-USE(OI)  then
            continue while ;
        else
            NON-HASH-USE(OI) := TRUE ;
        end if ;

    end case ;
```

```
        $ Propagate through assignment instructions.

        if IS~OVAR(OI) and OI~OP(OI) in OPS~ASN then
            WORK with [[P,IFROMO(OI,1)],USE] ;
        end if IS~OVAR ;

    end if IS~OVAR ;

    $ Propagate backward through BFROM chain

    WORK + { [[P,OI],USE] : [P,OI]∈BFROM(OI) ;
              | if USE=U~MEMB then not HASH~USE(OI)
                elseif USE=U~TYPE then not NON~HASH~USE(OI)} ;

end while ;
return ;

end proc USE~DETERM ;


proc BASING~PROP ;

$ This procedure propagates member basing pointers from
$ inserted locate instructions and value retrieval
$ instructions to other variable occurrences, wherever
$ necessary.  In order to allow multiple representations
$ for a variable occurrence, the map MODE is used as a
$ multi-valued map.

$ This routine is called by the routine UPDMODES.

$ The GLOBAL variables referenced by this routine include
```

```
$      BLOCKS       - set of code blocks
$      MODE{OI}     - modes of occurrence OI
$      ARG(I)       - arguments of instruction I

$ The macros used in this routine include
$      FORALLCODE(B,I) - for each instruction I in block B,
$                        - see (M4).
$      MGTYP(OI)    - gross type of occurrence OI, see (M26).
$      COMPTYP(M)   - element mode of mode descriptor M, see (M19).
$      DOMTYP(M)    - domain mode of mode descriptor M, see (M22).
$      RANTYP(M)    - range mode of mode descriptor M, see (M23).
$      CTYPN(M,I)   - I-th component mode of mode descriptor M,
                    - see (M20).
$      OI¬VALUE(OI) - value of occurrence OI, see (M9).

$ The local variables defined in this routine are

 repr
     I : int ;                      $ instruction identifier
     OV, IV1, IV2 : ∈OI¬BASE ;      $ variable occurrences
     DONE : set([∈OI¬BASE,∈MODE¬BASE]) ;
                                    $ occurrences which have been
                                    $ processed
     WORK : set([∈OI¬BASE,∈MODE¬BASE]) ;
                                    $ workpile of occurrences with
                                    $ member basings
     M, MD : ∈MODE¬BASE ;           $ mode descriptors
 end repr ;
```

```
WORK := NL ;            $ Initialize workpile.

$ For each instruction

( ∀ B∈BLOCKS,FORALLCODE(B,I) )

    [OV,IV1,IV2] := ARG(I) ;    $ Unpack arguments.

    $ If this instrution has no ovariable then continue
    $ processing next instruction.

    if OV = OM then continue ∀ ;

    $ At this point, the ovariables of inserted locate
    $ instructions will have been assigned member basing
    $ modes, and these are the only ovariable occurrences
    $ which have been assigned such modes (by MODE map).
    $ The mode map of all other ovariable occurrences will
    $ contain domain basings.

    M := OM ;

    if MGTYP(OV)=TELMT then

        $ Insert [OV,MODE(OI)] into workpile to indicate
        $ that OV has member basing pointer.

        WORK with [OV,MODE(OV)] ;

        $ If OV also needs domain basing, then OV is
        $ assigned multiple basings by absorbing the mode
        $ of its ivariable.
```

```
    if NON-HASH-USE(OV) then
        MODE(OV) + MODE(IV1) ;
    end if ;

else

$ Member basing pointers can also be transmitted by
$ value retrieval operations.

    case OPCODE(I) of

    (Q1-ARB,Q1-FROM,Q1-NEXT,Q1-INEXT) :

        if MGTYP(IV1) = TSET and ELMBASE(IV1) /= OM then

            $ For a value retrieval from a based set :

            M := COMPTYP(MODE(IV1)) ;
        end if ;

    (Q1-NEXTD,Q1-INEXTD) :

        if MGTYP(IV1) = TMAP and DOMBASE(IV1) /= OM then

            $ For an iteration over a based map :

            M := DOMTYP(MODE(IV1)) ;
        end if ;

    (Q1-OF) :

        case MGTYP(IV1) of

        (TMAP) :
```

```
            if DOMBASE(IV1) /= OM then

                $ For a value retrieval from a based map
                $ :

                M := RANTYP(MODE(IV1)) ;
            end if ;

        (THTUP) :

            if ELMBASE(IV1) /= OM then

                $ For a value retrieval from a based
                $ tuple of known length :

                M := COMPTYP(MODE(IV1)) ;
            end if ;

        (TMTUP) :

            if ELMBASE(IV1) /= OM then

                $ For a value retrieval from a based
                $ tuple of unknown length :

                M := CTYPN(MODE(IV1),OI-VALUE(IV2)) ;
            end if ;

        end case MGTYP ;

    end case OPCODE ;

    $ If M is not undefined then insert [OV,M] into
    $ workpile to indicate that member basing M is
```

```
        $ available at OV.

        if M /= OM then WORK with [OV,M] ; end if ;

    end if MGTYP ;

end ∀ ;

$ Propagate member basings through FFROM chain and
$ assignment instructions.

DONE := NL ;

$ Work contains the occurrences with member basings.

( while WORK /= NL )

    [OI,M] from WORK  ;

    $ Assign proper mode to OI according to HASH¬USE(OI)
    $ and NON¬HASH¬USE(OI).

    if HASH¬USE(OI) then

        if NON¬HASH¬USE(OI) then

            $ If both HASH¬USE(OI) and NON¬HASH¬USE(OI) are
            $ true, then assign OI member basing in
            $ addition to domain basing.

            MODE(OI) with M ;

        else
```

```
            $ If only HASH-USE(OI) is true, then assign OV
            $ member basing only.

            MODE(OI) :=  {M} ;

         end if NON-HASH-USE ;

        $ Indicate OI has been processed and insert the
        $ occurrences which are linked to OI and have not
        $ been processed into the workpile.

        DONE with [OI,M] ;
        WORK + { [WOI,M],[-,WOI]∈FFROM(OI) | [WOI,M] notin DONE }

        $ If OI is the ivariable of an assignment
        $ instruction, then insert the ovariable into work
        $ if it has not been processed.

       if OI-OP(OI) in OPS-ASN and IS-IVAR(OI)
           and [(OV := OFROMI(OI)),M] notin DONE then
             WORK with [OV,M] ;
        end if OI-OP ;
    end if ;HASH-USE ;

    $ Otherwise, NON-HASH-USE(OI) must be true and OI
    $ already has domain basing.  Nothing has to be done.

end while WORK ;

return ;

end BASING-PROP ;
```

```
proc REFINE ;

$ This procedure applies the heuristics explained in an
$ earlier chapter to choose between local, remote and
$ sparse representation for based objects.  These
$ heuristics amount to the following :

$ A) A based object should be sparse if it is to be iterated
$ over, unless we can show that the object is actually
$ identical with its base.  The routine ID-BASE will spot
$ such identities in a few potentially useful cases.

$ B) If no iteration is performed on an object, but it is
$ subject to algebraic operations (union, intersection,
$ etc) or is passed as a parameter, assigned and used
$ destructively, or inserted into a larger object, then it
$ should be remote.

$ C) If only differential updating operations are applied to
$ an object, and it is never transmitted to another by
$ assignment, insertion or call, then it can have a local
$ representation.

$ This routine is called by the main routine AUTO-DATA.  It
$ calls the routines ID-BASE and MAKE-REMOTE.  .

$ The global variables referenced by this routine include
$      LIVEPDS      - the set of live periods
$      MODE{OI}     - modes of occurrence OI

$ The macros used in this routine include
```

```
$      GROSSTYP(M) - gross type of mode M, see (M19).
$      REPRATT(M)  - representation attribute of domain basing M,
$                  - see (M25).
$      OI-OP(OI)   - opcode of the instruction which contains
$                  - occurrence OI, see (M7).

$ The local variables defined in this routine are

 repr
      LPD : ∈LPD-BASE ;            $ live period
      ATTRIB : ATOM ;              $ representation attribute
      OI : ∈OI-BASE ;              $ variable occurrence
      MD : ∈MODE-BASE ;            $ mode descriptors
end repr ;

$ Find the occurrences which are identical in value with
$ their bases.

ID-BASE();

$ For each live period of composite objects which are domain
$ based

(∀LPD ∈LIVEPDS | (∃MD∈MODE{arb LPD} | GROSSTYP(MD) in {TMAP,TSET}

    $ MD is the domain basing of a set or map.

    if MD = OM then continue ; ;

    $ If there is an occurrence in the live period subject
    $ to iteration operations and not identical with its
```

```
    $ base, then all the occurrences in the live period
    $ are assigned sparse representations.

    if (∃ OI ∈ LPD | (OI¬OP(OI) in {Q1¬NEXT, Q1¬NEXTD}
                and not ID¬TO¬BASE(OI))) then
        ATTRIB := SPARSE ;

    $ If any occurrence in the live period should have
    $ remote representation, all the occurrences in this
    $ live period are assigned remote representations.

    elseif ∃OI ∈ LPD | MAKE¬REMOTE(OI) then
        ATTRIB := REMOTE ;

    else
        ATTRIB := LOCAL ;

    end if ;

    $ Assign the calculated attribute to the mode of each
    $ occurrence in the live period.

    (∀OI ∈ LPD | (∃ MD∈MODE(OI) | GROSSTYP(MD) in {TMAP,TSET}) )
        REPRATT(MD) := ATTRIB ; ;

    end ∀OI ;
end ∀LPD ;

end proc REFINE ;


proc ID¬BASE() ;
```

```
$ This routine identifies a potentially important situation
$ in which the value of a variable occurrence is identical
$ (in value) with its base.  If all the elements X of a
$ base B are inserted into B by the operation of a set
$ former instruction which generates a set S, then S and B
$ will be identical collections.  This ID-TO-BASE property
$ can be propagated from S to other occurrences which are
$ linked and only linked to S.

$ This routine is called by the routine REFINE and calls the
$ routine SETOF.

$ The global variables referenced by this routine include
$     BASE-ELMTS(b)  - sets of occurrences which have been
$                    - known to be elements of base B
$     ID-TO-BASE(OI) - true if occurrence OI is
$                    - identical in value with its base
$     BFROM(OI)      - occurrences to which OI is directly
$                    - linked
$     FFROM(OI)      - occurrences which are directly
$                    - linked to OI

$ The macros used in this routine include
$     OFROMI(OI)   - the ovariable of the instruction
$                  - containing the ivariable OI
$     OI-OP(OI)    - opcode of the instruction containing OI,
$                  - see (M7).

$ The local variables defined in this routine are
```

```
 repr
      OI, WOI, SOI : ∈OI¬BASE ;   $ variable occurrences
      WORK : set(∈OI¬BASE) ;      $ workpile of occurrences
 end repr ;

$ For each base whose elements are inserted at only one
$ place

(∀ BASE ∈ DOMAIN BASE¬ELMTS | #BASE¬ELMT{BASE} = 1 )

    $ Find the variable occurrence inserted into the base.

    WOI := arb BASE¬ELMTS{BASE} ;

    $ If WOI is not an argument of a set former, bypass the
    $ base.  The value SOI returned from the routine SETOF
    $ is the set being constructed if WOI is an argument
    $ of a set former, otherwise undefined.

    if (SOI := SETOF(WOI)) = OM then continue ∀ ;

    $ Otherwise, the ovariable of the set former instruction
    $ is identical in value to its base.

    ID¬TO¬BASE(SOI) := TRUE ;

    $ Propagate the property ID¬TO¬BASE of SOI to the
    $ occurrences which are linked to SOI and only to SOI.

    WORK := {OI, [-,OI]∈FFROM{SOI} | #BFROM{OI}=1 } ;

    ( while WORK /= NL )
```

```
        OI from WORK ;
        ID-TO-BASE(OI) := TRUE ;

        $ Propagate through assignment instructions.

        if OI-OP(OI) in OPS-ASN and IS-IVAR(OI)
                    and not ID-TO-BASE(OV:=OFROMI(OI)) then
            WORK with OV ;;

        $ Propagate through FFROM chain.

        WORK := {WOI, [-,WOI]∈FFROM(OI) | #BFROM(WOI)=1
                        and not ID-TO-BASE(OI) } ;
    end while ;
end ∀ BASE ;

return ;
end proc ID-BASE ;


proc SETOF(OI) ;

$ This routine returns OM unless OI is an argument of a set
$ former instruction.  In this case, the set generated by
$ the instruction is returned.  Since set former
$ instructions in SETL source programs will have been
$ expanded into series of instructions, a number of
$ instructions must be examined to detect set formers.  This
$ routine is therefore compiler dependent and must be
$ updated whenever the compiler is modified.  If the
$ compiler could leave an explicit syntatic indication of
```

```
$ set former instruction, this routine could then be
$ replaced by a macro which simply detects the indicator.

$ This routine is called by the routine ID-BASE.

$ The global variables referenced by this routine include
$      BFROM{OI}    - occurrences to which OI is directly
$                   - linked
$      FFROM{OI}    - occurrences which are directly
$                   - linked to OI

$ The macros used in this routine include
$      OFROMI(OI)   - the ovariable of the instruction
$                   - containing the ivariable OI

$ The local variables defined in this routine are

 repr
      OI, WOI, NOI : ∈OI-BASE ;   $ variable occurrences
      WORK : set(∈OI-BASE) ;      $ temporary set of occurrences
 end repr ;

$ If OI is an argument of a set former then it will link to
$ and only to an occurrence which is pushed into a stack
$ which is then subject to a Ω1-SET1 operation.  The
$ following code detects this case.

$ If OI links to more than one occurrence we do not have the
$ configuration that we are looking for.

if #(WORK := {WOI, [-,WOI]∈FFROM{OI} | #BFROM{OI}=1 } ) = 1 then
```

```
    $ WOI is the occurrence linked to OI.

    WOI := arb WORK ;

    $ WOI should be pushed into a stack.

    if OI¬OP(WOI)/=Q1¬PUSH then return OM ;
    else

        $ NOI is the stack.

        NOI := OFROMI(WOI) ;
        if #(WORK:={WOI,[-,WOI]∈FFROM{OI} | #BFROM{OI}=1})=1 then

            $ The stack should be subject to a Q1¬SET1
            $ operation.

            WOI := arb WORK ;
            if OI¬OP(WOI) /= Q1∈SET1 then return OM ;
               else return OFROMI(WOI) ;
            end if OI¬OP ;

        else
            return OM ;
        end if # ;

    end if OI¬OP ;
else

    return OM ;
end if # ;

end proc SETOF ;
```

```
proc MAKE-REMOTE(OI) ;

$ This boolean procedure determines whether an occurrence OI
$ should have a remote representation, by detecting its
$ appearance as an argument of operations which are
$ particularly inefficient for local representations.

$ This routine is called by the routine REFINE.

$ The global variables referenced by this routine include
$      COPY-FLAG(OI)  - true if OI cannot be used destructively

$ The macros used in this routine include
$      OI-OP(OI)    - opcode of the instruction which contains
$                   - occurrence OI, see (m7).
$      ARGNO(OI)    - argument number of occurrence OI, see (M6).

$ The local variables defined in this routine are
 repr
      OI : ∈OI-BASE ;              $ variable occurrence
      OP : ∈OPCODES ;              $ opcode
 end repr ;

OP := OI-OP(OI) ;

$ If OI is an argument of an algebraic or boolean operation
if ( OP ∈ {Q1-ADD, Q1-SUB, Q1-MULT, Q1-MOD, Q1-INC, Q1-EQ, Q1-NE}

     $ Or OI is the ovariable of a retrieval operation
     or ( ARGNO(OI)=1 and OP in {Q1-OF, Q1-ARB, Q1-FROM} )
```

```
    $ Or OI is the input argument of a set of tuple former
    or ( ARGNO(OI)=2 and OP in {Q1~PUSH, Q1~POP} )

    $ Or OI is the input argument of an incorporation
    $ operation
    or ( ARGNO(OI)=3 and OP in {Q1~SOF, Q1~WITH} )

    $ Or OI cannot be used destructively
    or COPY~FLAG(OI)

    $ Or OI is formal parameter or an actual argument of a
    $ procedure,
    or ( OP in {Q1~ARGIN, Q1~ARGOUT} )

then
    $ Then OI should have remote representation.
    return TRUE;
else
      return FALSE ;
end if ;

end proc MAKE~REMOTE ;


end AUTO~DSTRUCT ;
```

APPENDIX A : PRIMITIVE SETL OPERATIONS

Operation	Remarks
X + Y	integer and real addition, set union, character string and tuple concatenation
X - Y	integer and real subtraction, set difference
X * Y	integer and real multiplication, set intersection, tuple and character string repetition
X / Y	integer and real division, set symmetric difference
X // Y	arithmetic remainder function
S ** Y	arithmetic exponentiation
X and Y	boolean and
X or Y	boolean or
X implies Y	boolean implies
not X	boolean negation
S = Y	equality comparison
X /= Y	inequality comparison
X > y, X < Y etc.	arithmetic comparisons
X := Y	simple assignment
#x	cardinality of set, lenth of tuple and string
arb X	select arbitrary element of set
X with Y	set extension, equivalent to X + {Y}
X less Y	set contraction, equivalent to X - {Y}
X in Y, X notin Y	set membership tests

Operation Remarks

X incs Y
: set inclusion test

pow(X)
: power set

{X,Y,...}
: set with specified elements

[x,y,...]
: tuple with specified elements

F(X)
: function or subprocedure call, indexing to component of tuple or string. If F is a map, F(X) is the unique Y such that [X,Y] in F, if such exists ; otherwise F(X) is undefined.

F(X1,..,Xn)
: function or subprocedure call. If F is a set, F(X1,..,Xn) is the unique Y such that [X1,..,Xn] in F, if such exists ; otherwise F(X1,..,Xn) is undefined.

F{X1,..Xn}
: F must be a map ; F{X1,..,Xn} is the set of all Y such that [X1,..,Xn,Y] in F.

F[X1,..,Xn]
: F must be a map ; F[X1,..Xn] is the set of all Y for which there exist Z1,..Zn with Z1 in X1, ...,Zn in Xn and [Z1,..,Zn,Y] in F.

F(x:y)
: extract subpart of length Y starting at component X of string or tuple F.

F(X):=Y
: assignment operation corresponding to retrieval operator F(X).

F(X1,..Xn) := Y ;

assignment operator corresponding to retrieval operator F(X1,..,Xn).

APPENDIX B : ALPHABETICAL LISTING OF GLOBAL NAMES REFERENCED

ALL-I - variable, see (V10).

ALL-O - variable, see (V9).

ALL-OI - variable, see (V8).

ARGNO - macro, see (M6).

ARGS - variable, see (V22).

ARG1 - macro, see (M1).

ARG2 - macro, see (M2).

ARG3 - macro, see (M3).

BASE-ELMT - variable, see (V27).

BASENAM - macro, see (M24).

BASING-PROP - procedure, see (P24).

BFROM - variable, see (V11).

BLOCK-BASE - base, see (V17).

BLOCKOF - variable, see (V20).

BMODE - variable, see (V29).

CAN-DROP - macro, see (M32).

COMBASE - macro, see (M31).

COMPTYP - macro, see (M19).

CONSTR-PS-CRTHIS - procedure, see (P2).

CTYPN - macro, see (M20).

DOMBASE - macro, see (M29).

DOMTYP - macro, see (M22).

ELMBASE - macro, see (M28).

EQUIV - procedure, see (P14).

ERROR-PATH - constant, see (C8).

FFROM - variable, see (V12).

FORALLCODE - macro, see (M4).

GENBASE - procedure, see (P1).

GENLOCS - procedure, see (P3).

GLTYP - macro, see (M27).

GMAP - constant, see (C20).

GROSSTYP - macro, see (M18).

GSET - constant, see (C21).

HASH-USE - variable, see (V33).

ID-BASE - procedure, see (P26).

ID-TO-BASE - variable, see (V35).

IFROMO - macro, see (M14).

INSERTLOCS - procedure, see (P13).

INSTNO - macro, see (M5).

INSTRS - base, see (V16).

IS-CONST - variable, see (V4).

IS-FORMAL - variable, see (V24).

IS-GLOB - variable, see (V5).

IS-HASHED - macro, see (M13).

IS-IVAR - macro, see (M12).

IS-OVAR - macro, see (M11).

IS-PRIM - macro, see (M17).

KNT - constant, see (C18).

LASTCALL - procedure, see (P18).

LENTYP - macro, see (M21).

LIVEPDS - variable, see (V32).

LOCAL - constant, see (C41).

LPD-BASE - base, see (V31).

MAKE-REMOTE - procedure, see (P28).

MAPSET - constant, see (C38).

MAPTUP - constant, see (C36).

MERGE - procedure, see (P11).

MERGE-INTO - procedure, see (P12).

MERGEOBJ - procedure, see (P4).

MGTYP - macro, see (M26).

MODE - variable, see (V30).

MODE-BASE - base, see (V28).

MODECMPRS - procedure, see (P21).

MODEDIS - procedure, see (P16).

MOVELOCS - procedure, see (P19).

NAME - variable, see (V2).

NBASEDON - variable, see (V26).

NBASES - variable, see (V25).

NEXT - variable, see (V19).

NON-HASH-USE - variable, see (V34).

NULL-PATH - constant, see (C7).

OFROMI - macro, see (M15).

OI-BASE - base, see (V6).

OI-INTOV - macro, see (M10).

OI-NAME - macro, see (M8).

OI-OP - macro, see (M7).

OI-VALUE - macro, see (M9).

OPCODE - variable, see (V21).

OPCODES - base, see (V18).

OPS-ASN - constant, see (C1).

OPS-CREATE - constant, see (C4).

OPS-HASH - constant, see (C2).

OPS-RETRIEVE - constant, see (C3).

PARTITION - procedure, see (P17).

PROPELMT - procedure, see (P5).

PROPOFMAP - procedure, see (P6).

PROPOFTUP - procedure, see (P8).

PROPSOFAMAP - procedure, see (P10).

PROPSOFMAP - procedure, see (P7).

PROPSOFTUP - procedure, see (P9).

PS-CRTHIS - variable, see (V13).

RANBASE - macro, see (M30).

RANTYP - macro, see (M23).

RC-BASE - base, see (V7).

RC-CALL - constant, see (C5).

RC-RETN - constant, see (C6).

REALB - procedure, see (P15).

REFINE - procedure, see (P25).

REMOTE - constant, see (C40).

REPRATT - macro, see (M25).

SB-BASE - base, see (V23).

SETOF - procedure, see (P27).

SETTUP - constant, see (C37).

SPARSE - constant, see (C39).

STRUCTPART - macro, see (M16).

SUBSTMD - procedure, see (P22).

SYMBOLS - base, see (V1).

TA - constant, see (C15).

TBASE - constant, see (C23).

TC - constant, see (C24).

TELMT - constant, see (C22).

TG - constant, see (C32).

THTUP - constant, see (C28).

TI - constant, see (C25).

TL - constant, see (C16).

TLC - constant, see (C14).

TLI - constant, see (C11).

TMAP - constant, see (C31).

TMTUP - constant, see (C27).

TNUM - constant, see (C26).

TOM - constant, see (C9).

TP - constant, see (C17).

TR - constant, see (C12).

TSC - constant, see (C13).

TSET - constant, see (C30).

TSI - constant, see (C10).

TSTRUCT - constant, see (C34).

TTUP - constant, see (C29).

TYPE¬BASE - base, see (V14).

TYPES - variable, see (V15).

TZ - constant, see (C33).

TZSTRUCT - constant, see (C35).

UNT - constant, see (C19).

UPDMODES - procedure, see (P20).

USE-DETERM - procedure, see (P23).

VALUE - variable, see (V3).

BIBLIOGRAPHY

Allen,F.E. [1972]

A Catalog of Optimizing Transformations, in

Design and Optimization of Compilers (R. Rustin ed.)

Prentice Hall publishing co.

Englewood Cliffs, N.J.

Allen,F.E. [1974]

Interprocedural Data Flow Analysis

Proc. IFIP conference 74

North Holland publishing co.

Amsterdam, Holland

Brent,R. [1973]

Reducing the Retrieval Time of Scatter Storage Techniques

CACM, Vol.19, No.2

Bruce.J. [1976]

An APL Optimizer

Ph.D. thesis

M.I.T.

Cambridge, Mass.

Cocke,J. and Schwartz,J.T. [1970]

Programming Languages and Their Compilers

Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Codd,E. [1970]

A Relational Model of Data for Large Shared Data Bank

CACM, Vol.13, No.6

Crick,M. and Symonds,A. [1970]

A Software Associative Memory for Complex Data Structures

IBM Cambridge Sci. Center.

Research report No. G320-2060

Derksen,J. [1972]

The QA4 Primer

Stanford Research Inst.

Stanford, Calif.

Dewar,R. [1977a]

Copy Optimization in SETL

SETL newsletter No. 164

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Dewar,R., Grand,A., Schwartz,J.T. and Schonberg,E. [1977b]

SETL Data Structures

SETL newsletter No. 189

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Dewar,R. [1978]

The SETL Programming Language

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

in preparation

Earley,J. [1971]

Toward an Understanding of Data Structures

CACM Vol.14, No.10

Earley,J. [1973a]

ReLational Level Data Structures for Programming languages

Comp. Sci. Dept.

Univ. of Calif., Berkeley.

Berkeley, Calif.

Earley,J. [1973b]

An Overview of the VERS2 Project

Memo ERL-M416

Electronics research laboratory, college of engineering,

Univ. of Calif, Berkeley.

Berkeley, Calif.

Earley,J. [1974a]

High Level Iterators and a Method of Automatically Designing

Data Structure Representations

Electronics research laboratory, college of engineering,

Univ. of Calif., Berkeley.

Berkeley, Calif.

Earley,J. [1974b]

High Level Operations in Automatic Programming

ACM, SIGPLAN notices, Vol.9, No.4

Feldman,J. [1972]

Automatic Programming

Tech. Report CS-255,

Comp. Sci. Dept., Standford Univ.

Stanford, Calif.

Grand,A. [1978]

The SETL Optimizer

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

in preparation

Guttag,J.V. [1975]

The Specification and Application to Programming of Abstract Data Types

Tech. Report CSRG-59

Dept. of Comp. Sci.

Univ. of Toronto

Toronto, Canada

Kant,E. [1977]

The Selection of Efficient Implementations for a High-Level Language

ACM, SIGPLAN Vol.12, No.8

August, 1977.

Kenendy,K. and Schwartz,J.T. [1975]

An Introduction to the Set Theoretical Language SETL

Jour. computers and mathematics with application

Vol.1, pp.87-119

Knuth,D. [1968]

Fundamental Algorithms: the Art of Computer Programming Vol 1

Addison-Wesley publishing co.

Reading, Mass.

Knuth,D. [1973]

Sorting and Searching: the Art of Computer Programming Vol 3

Addison-Wesley publishing co.

Reading, Mass.

Liu,S.C. and Schonberg,E. [1977]

Uncovering Profitable Basing Relations

SETL newsletter No. 180

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Low,J.R. [1974]

Automatic Coding : Choice of Data Structures

Ph.D. thesis

Comp. Sci. Dept. Memo STAN-CS-74-452

Stanford Univ.

Stanford, Calif.

Low,J.R. [1978]

Automatic Data Structure Selection : An Example and Overview

CACM Vol.21, No.5

Mcdermott,D. and Sussman G. [1972]

The CONNIVER Reference Manual

AI memo No. 259

M.I.T.

Cambridge, Mass.

Morris,J. [1973]

A Comparison of MADCAP and SETL

Univ. of Calif., Los Alamos Sci. Lab.

Stanford, Calif.

Rovner,P. [1976]

Automatic Representation Selection for Associative Data Structures

Comp. Sci. Dept. TR10

Univ. of Rochester

Rochester, N.Y.

Schaefer,M. [1973]

A Mathematical Therory of Global Program Optimization

Prentice Hall publishing co.

Englewood cliffs, N.J.

Schonberg,E., Dewar,R., Vanek,L. and Grand,A. [1976]

Some Changes to the SETL Language in Preparation for the OptImizer implementation

SETL newletter 169

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Schonberg,E. and Liu,S.C. [1977]

Manual and Automatic Data Structuring in SETL
Implementation and Design of Algorithmic Languages
Proceedings of the 5th Annual iii Conference
Guidel, France

Schwartz,J.T. [1971a]
An Additional Preliminary Remark on the Importance of 'Object Type' for SETL, with Some Reflections on the Motion of 'Data Structure Language'
SETL newletter 31
Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.
New York, N.Y.

Schwartz,J.T. [1971b]
More Detailed Suggestions Concerning 'Data Strategy'
SETL newletter 39
Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.
New York, N.Y.

Schwartz,J.T. [1973]
On Programming: an Interim Report on the SETL Project.
Installment 1: Generalities.
Installment 2: The SETL Language and Examples of Its Use.
Courant Inst. Math. Sci., New York Univ.
New York, N.Y.

Schwartz,J.T. [1974a]
Automatic and Semiautomatic Optimization of SETL.
ACM, SIGPLAN notices, Vol.9, No. 4, pp. 43 ff.

Schwartz,J.T. and Paige,R. [1974b]

On Jay Earley's Method of Iterator Inversion

SETL newsletter 138

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Schwartz,J.T. [1975a]

Automatic Data Structure Choice in a Language of Very High Level

CACM, Vol.18, No.12, pp. 722-728

Schwartz,J.T. [1975b]

Optimization Of Very High Level Languages - I :

Value Transmission and Its Corollaries

Journal of computer languages

Vol.1, No.1, pp. 161-194

Schwartz,J.T. [1975c]

Optimization Of Very High Level Languages - II :

Deducing Relationships of Inclusion and Membership

Journal of computer languages

Vol.1, No.2, pp. 197-218

Schwartz,J.T., Dewar,R. and Schonberg,E. [1976a]

More on Basings

SETL newletter 171.

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Schwartz,J.T. and Dewar,R. [1976b]

Basing Semantics Revisited

SETL newletter 171a.

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Sharir,M. [1977]

An Algorithm for Copy Optimization

SETL newsletter No. 195

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Sharir,M. [1978a]

Simplified Approach to Automatic Data STruture Choice

SETL newsletter No. 203

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Sharir,M. [1978b]

Second Simplified Approach to Automatic Data Structure Choice

SETL newsletter No. 207

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Sussman,G., Winograd,T. and Charniak,E. [1970]

Micro-PLANNER Reference Manual

AI memo No.9 203, project MAC

M.I.T.

Cambridge, Mass.

Tenenbaum,A.M. [1974]

Type Determination for Very High Level Languages.
Ph.D. thesis
Research rep. NSO-3
Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.
New York, N.Y.

Tompa,F.W. [1974]
Evaluating the Efficiency of Storage Structures
Research report CS-75-16
Dept. of Comp. Sci.
Univ. of waterloo

Tsui,w.h. [1977]
A Reformulation of Value-Flow Analysis
SETL newsletter No. 181
Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.
New York, N.Y.

Ullman,J.D. [1975]
Data Flow Analysis
Tech. Rep. 179
Dept. of Elec. Eng., Comp. Sci. Lab.,
Princeton Univ.
Princeton, N.J.

VanLehn,K. [1973]
SAIL User Manual
Stanford Comp. Sci. Tech. Report STAN-CS-93-373.
Stanford Univ.

Stanford, Calif.

Vanek,L [1976a]

Simplifying and Extending the SETL Type Calculus

SETL newletter 173.

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Vanek,L [1976b]

Global Analysis Techniques for the Optimizing SETL Compiler

Proceedings of the September 1976 MOSCOW conference on very High Level Languages

Comp. Sci. Dept., Courant Inst. Math. Sci., New York Univ.

New York, N.Y.

Wegbreit,B. [1973]

The Treatment of Data Types in EL1

Harvard Univ.

Cambridge, Mass.

Zilles,S.N. [1975]

Data Algebra : a Specification Technique for Data Structure

Ph.D. thesis

EXPRESSION CONTINUITY AND THE FORMAL DIFFERENTATION OF ALGORITHMS

Robert Paige

CONTENTS

		Page
PREFACE		270
I.	INTRODUCTION	272
II.	BASIC CONCEPTS AND EXAMPLES	285
	A. Preliminaries	285
	B. Initial Examples	286
	C. Formal Differentiation of Set Theoretical Expressions Continuous in All of Their Parameters	293
	D. Differentiation of Expressions Containing Parameters on which They Depend Discontinuously	315
III.	AN IMPLEMENTATION DESIGN OF FORMAL DIFFERENTIATION OF EXPRESSIONS CONTINUOUS IN ALL OF THEIR PARAMETERS	364
	A. Overview and System Design	364
	B. System Description	368
	C. Computing the Formal Derivative	386
IV.	IMPLEMENTATION DESIGN FOR FORMAL DIFFERENTIATION OF EXPRESSIONS CONTINUOUS IN SOME OF THEIR PARAMETERS	422
	A. Introduction	422
	B. Semiautomatic Formal Differentiation of Discontinuous Expressions	423
	C. Implementation Design for SETL	453
	D. Applications to Algorithm Derivation	464
	E. Conclusion	491
APPENDIX		
	A. SETL and SUBSETL	521
	B. Predicting Speedup for Rule 1	536
	C. Formal Differentiation Tables	538
	D. Various Elementary and Compound Set Theoretic Transformations	575
	E. Assorted Utility Routines for a Source to Source Transformational Implementation	599
	F. Additional Case Studies	619
BIBLIOGRAPHY		654

PREFACE

Abstract

Formal differentiation is a program optimization technique which generalizes John Cocke's method of strength reduction and provides a convenient framework with which to implement a host of program transformations including Jay Earley's 'iterator inversion'. This technique captures a commonly occurring yet distinctive mechanism of program construction in which succinct algorithms involving costly repeated calculations are transformed into more efficient incremental versions.

The basic formal idea of this technique can be put as follows. Suppose that an expression $C = f(x_1,\ldots,x_n)$ will be used repeatedly in a a program region R, but that its calculation cannot be moved outside R because its parameters $x_1,\ldots,x_n$ are modified within R. If we make C available on each entry to R (by calculating it before entry) and keep C available within R by recalculating it each time one of its parameters is modified, then we may be able to avoid all full calculations of C within R.

For this approach to be reasonable we require the following heuristic condition to hold: Within R, for each redefinition $x_j = \Delta_{x_j}$ to a parameter x_j, there should exist code which can be inserted immediately before and after the redefinition point p and which serves to keep C available within R. We refer to this inserted code as the *pre* and *post derivative* code of f with respect to the change $x_j = \Delta_{x_j}$. We require this code as well as all code necessary to maintain available expressions on which it depends to consist of "easy" calculations relative to the cost of a fresh calculation of f (i.e. operations heuristically less costly in time). If this is the case, then we say (suggestively though only heuristically and, of course, not in the standard technical sense) that the expression f is continuous in its parameters relative to the modifications occurring within R. If we cannot demonstrate that f is continuous with respect to a particular parameter change, we will say that f is discontinuous in that parameter change.

Up until Earley's discovery of iterator inversion the preceding idea was applied at the Fortran level for expressions continuous in all of their parameters. Application of this idea in a set theoretic context was introduced by Jay Earley with his discovery of 'iterator inversion'. Moreover, Earley's transformations could handle expressions involved in discontinuity parameters. However, Earley lacked an implementation design.

In the present paper we unify the technique of Cocke and Earley, and provide algorithms which can implement these formal differentiation transformations both automatically and semiautomatically for programming languages ranging from Fortran to SETL. However, we find that success is best achieved in the case of SETL. Thus, we study set theoretic formal differentiation in depth and present a comprehensive semiautomatic implementation design for a restricted version of SETL called Subsetl. We show that the expected speedup due to

transformations applied by our proposed system can be as great as an order of magnitude. In particular we regard differentiation of general set formers

$C = \{x \in s \mid K(x,t_1,\ldots,t_n)\}$ as being of primary importance. We estimate that the cost of executing a calculation C repeatedly in a loop L is proportional to $N \times (\#S) \times Cost(K)$ where N is the iteration count of L. The formal differentiation transformations applied by our system will keep the value of C available in either $(N + \#S) \times Cost(K)$ or $(N + (\#S) \times \log(\#S)) \times Cost(K)$ elementary steps; and this will usually imply a speedup.

We illustrate our proposed system by considering and improving eight sample Subsetl programs. We feel that these initial case studies lend strong support to further efforts to fully automate and incorporate set theoretic formal differentiation as part of an optimizing compiler.

Acknowledgements

I thank all the people of the Courant Institute who read through sections of my original draft of this paper and offered helpful corrections and improvements. In this regard I appreciate the attention of Janet Fabri, Shaye Koenig, Ed Schoenberg, and Len Vanek. I am especially grateful to my advisor, Jack Schwartz, who made important initial contributions to my work, guided me along the way, and offered valuable recommendations for improving my expository style. Finally, I thank Connie Engle, the extraordinary typist of the Courant Institute who typed my paper with the same care that she has sent so many other manuscripts to press.

I. INTRODUCTION

Continued development of very high level languages depends in part on our ability to recognize common major aspects of programming style as resulting from the application of some standard technique of program improvement to an underlying program prototype. A technique of program improvement that we are able to perceive as general can become the basis for a general optimization method; and once this method is in hand, we can safely write programs in relatively simple unoptimized forms, since their more complex optimized forms will be seen as obvious improvements, derivable mechanically or semimechanically from these simple forms.

This thesis describes an optimization method, formal differentiation, which generalizes the classical method of 'reduction in operator strength'. The method captures a commonly occuring yet distinctive mechanism of program construction in which succinct algorithms involving costly repeated calculations are transformed into more efficient incremental versions. When applied to set theoretic dictions as found in a language such as SETL (cf. Appendix A for SETL description), this technique can transform algorithms from high level concise but inefficient problem statements into more complex but efficient program versions.

Much of our work is based on Jay Earley's technique of 'iterator inversion' which was applied in the set theoretic context of his proposed language VERS2 [E1,E2]. The current

thesis will view 'iterator inversion' along with its generalizations as a kind of 'formal differentiation' of algorithms. Although the idea of formal differentiation can be applied independently of semantic level, it is particularly well suited to very high level languages. Thus, we will describe an implementation design for an interactive semiautomatic system which would facilitate the application of this technique to algorithms written in SETL. We give pragmatic rules for the recognition and treatment of reasonably general cases in which this optimization is applicable, and consider some of the problems which arise in actually attempting to install this optimization as part of a compiling system.

Before presenting a formal description of our method, we trace through its origins. Historically, similar transformations have been used in numerical techniques to tabulate values of mathematical functions; e.g., using the compound growth formula,

$$f(t) = P * (1 + I)^t \tag{1}$$

for a given initial value P and growth rate I, we can calculate the sequence f(0),f(1),f(2),... by first computing f(0) = P and then generating each successive entry f(t) from the previous entry f(t - 1) by applying the identity f(t) = f(t - 1) * (1 + I). The computational cost of this method is usually much less than the cost of repeated calculations of (1).

Charles Babbage's analytic difference engine further illustrates this idea [Gl]. Babbage's early computer could perform only one arithmetical operation, addition, but by use of difference polynomials he could program his machine to calculate tables for polynomials efficiently. We can see how to do this by noting that for a given polynomial p(x) of degree n and an increment Δ , the first difference polynomial $p_1(x) = p(x + \Delta) - p(x)$ is of degree n-1 or less, the second difference polynomial $p_2(x) = p_1(x + \Delta) - p_1(x)$ is of degree n-2 or less, ..., and finally $p_n(x)$ must be a constant. Thus, in order to generate a sequence of polynomial values $p(x_0)$, $p(x_0 + \Delta)$, $p(x_0 + 2\Delta)$, ..., we can do the following:

1. Calculate initial values for $p(x_0), p_1(x_0), \ldots p_n(x_0)$ which are stored in components t(1),t(2),...,t(n+1) of an n+1-tuple t.

2. Generate the desired polynomial table by iterating over the code block below,

```
PRINT x, t(1);          /* PRINT x and p(x) */
x := x + Δ;             /* increment x */
t(1) := t(1)+t(2);      /* place new values for */
t(2) := t(2)+t(3);      /* p(x),p1(x),...,pn-1(x) into */
    :
t(n) := t(n)+t(n+1); /* t(1),t(2),...,t(n) */
```

In the 1960's John Cocke emphasized the general significance of an optimization method he called 'reduction in operator strength' which incorporates the ideas just mentioned and applies them to Fortran level code [Sch7].

Cocke's original techniques have since been generalized and implemented with various improvements (for which see A1, C1, C2, K1-K5). We illustrate his method with the following simple example. Suppose that an expression i * c occurring in a strongly connected region R cannot be moved out of R because of redefinitions to i. (We assume here that c is a region constant of R.) Suppose also that the variable i is defined before each entry to R and that all redefinitions to i within R are of the form i = i $\pm$ Δ where Δ is a region constant of R. Then we can use the following idea to move all calculations of i * c out of R. Since i is defined on entrance to R, we can insert an assignment T = i * c to a unique compiler generated variable T just prior to each entry point of R. Within R immediately before each redefinition i = i $\pm$ Δ to i we can preserve the value of i * c in T by executing the update assignment T = T $\pm$ Δ * c (whose form follows from the distributive law). Note that Δ * c is invariant, and its calculation can be moved out of R. Finally, we see that all calculations of i * c are redundant in R and can be replaced by uses of T.

A slightly more complicated example will serve to illustrate the deeper problems which can arise in applying reduction in strength when data values must be traced from one variable to another. Consider the last example once more, but now permit redefinitions to i within R of the forms i = - k, i = - k $\pm$ ℓ and i = k $\pm$ ℓ where k and ℓ are either region constants or variables. In such a case

we can sometimes still reduce the multiplication i * c to successive additions and copy operations. After inserting the initial assignments T = i * c, as before, we can keep the value of i * c current in T by inserting the following code just prior to each redefinition to i,

redefinition	update code
i = - k	T = - k * c
i = - k $\pm$ ℓ	T = - k * c $\pm$ ℓ * c
i = k $\pm$ ℓ	T = k * c $\pm$ ℓ * c

Any product i * c or c * i introduced as part of the update code can be replaced by T. All region constant products can be moved out of R. Common subexpressions introduced by this transformation can be eliminated. Finally, for i * c to be reduced in strength, all remaining products occurring within the update code must be reducible in strength. If this condition is satisfied and if the time cost of the addition operations inserted into R by strength reduction is less than the cost of the multiplications i * c removed from the original text, then a constant factor improvement in running time should be obtained.

The program transformations described above represent a most basic kind of 'formal differentiation' which we believe is a term more descriptive of the process being applied than Cocke's term 'reduction in strength'. Note, also, that the phrase, 'reduction in strength' has been applied to other optimization techniques which replace a costly operation with a less expensive one (e.g., peephole

optimizations such as the string concatenation removal length(STRING1 ‖ STRING2) ⇒ length(STRING1) + length(STRING2)).

For systematic discussion of the notion of formal differentiation, it is convenient to introduce some definitions and notational devices. We will sometimes use the notation $C = f(x_1,\ldots,x_n)$ to associate a text expression f with a unique compiler generated variable C. We will assume that whenever f is executed, its value calculated from the values of its free variables $x_1,\ldots,x_n$ and constants, is placed in C. We will say that C is *available on exit* from a program point p if C is equal to the value which the expression f would have if evaluated immediately after the statement at p is executed; C is *available on entrance* to p if C is available on exit from all predecessor points of p. If C is available on entrance to p, and if C is not available on exit from p (which will happen when execution of the statement at p changes the value of a parameter x_i upon which the value of f depends), then we say that C is *spoiled* at p. C is available on entrance to a program region R if it is available on entrance to each entry point of R.

As Schwartz notes in [C2], reduction in strength (which we shall call formal differentiation) is a very general and powerful optimization method applicable to a wide class of operations. The basic formal idea of this technique can be put as follows. Suppose that an expression $C = f(x_1,\ldots,x_n)$ will be used repeatedly in a program region R, but that its calculation cannot be moved outside R because its parameters

$x_1,\ldots,x_n$ are modified within R. If we make C available on each entry to R (by calculating it before entry) and keep C available within R by recalculating it each time one of its parameters is modified, then we may be able to avoid all full calculations of C within R.

For this approach to be reasonable, there must be some way of *recalculating* C more easily after its parameters are modified than by calculating C afresh each time it is required. For this to be the case, we are likely to require three conditions which can be stated heuristically as follows:
(a) Each free variable $x_1,\ldots,x_n$ on which f depends must be defined on entrance to R. This insures that initial calculations to make C available on entrance to R are possible.
(b) Within R, for each redefinition $x_j = \Delta_{x_j}$ to a parameter x_j, there should exist code which can be inserted immediately before and after the redefinition point p and which serves to keep C available within R (except possibly at points of the update code itself). We refer to this inserted code as the pre and post derivative code of f with respect to the change $x_j := \Delta_{x_j}$. We require this code as well as all code necessary to maintain available expressions on which it depends to consist of 'easy' calculations relative to f (i.e. operations heuristically less costly in time). If this is the case, then we say (suggestively though only heuristically and, of course, not in the standard technical sense) that the expression f is continuous in its parameters relative to the modifications occuring within R. If we cannot

demonstrate that f is continuous with respect to a particular parameter change, we will say that f is discontinuous in that parameter change.

(c) No modification to an argument of $f(x_1,\ldots,x_n)$ may occur in a strongly connected subregion Q of R since it is likely that any update code inserted into Q would be executed much more frequently than at program points in R - Q, and hence could cost much more in running time than the original calculation $f(x_1,\ldots,x_n)$.

An implementation method for formal differentiation must include the following steps: 1. find reduction candidates; 2. test for the enabling conditions above; 3. provide rules for generating pre and post derivatives; 4. transform the original code by successive applications of the strength reduction method, possibly in several ways; 5. select the most profitable of the transformed program versions.

Using the general framework above it is possible to methodically reduce classes of expressions built up from rather complicated operations and data.

Although Earley was the first to describe formal differentiation in a set theoretic context [E1], an implementation design for set expressions was first provided by Fong and Ullman [F1,F2]. They propose to reduce binary set operations such as union, intersection, and set difference as well as more complicated expressions. They provide a straightforward mechanism for detecting and reducing such expressions, and within their model they predict possible program speedup by

as much as an order of magnitude, while also guaranteeing a constant bounded space and work overhead. Instead of maintaining the full value of a candidate expression within a program region R, their method keeps the small variation in value of this expression available in R. They emphasize that aside from the nature of specific operations, the control flow structure is an important factor in determining reduction in strength capabilities. Drawing on control flow considerations, they demonstrate that certain 'induction' variables on which reduction candidate expressions depend may be redefined in R by reassignment to 'induction' expressions as well as by differential modifications.

However, the work of Jay Earley is of central importance to the present thesis. In [E1] he describes an interesting data choice optimization for the high level set theoretical language VERS2 [E2]. Earley notes that his optimizations will apply to any language of about the same level as VERS2, and since SETL is roughly of this level, SETL dictions define a convenient context for study of the Earley optimizations, cf. [E1,S1,Schl-6]. Earley calls his optimization method 'iterator inversion', and thinks of it as a way of automatically choosing a representation of a set or sequence being iterated over, which minimizes the number of operations necessary for a VERS2 iterator to produce its stream of values. Ideally, iterator inversion will produce an iterator's stream of values (in the correct order) directly. For example, an iterator $x \in s \mid K(x)$ produces the stream

of all elements in the set s satisfying the Boolean sub-expression K(x). Iterator inversion, in Earley's sense, will replace an iterator $x \in s \mid K(x)$ by a simpler iterator $x \in s'$ where s' represents the set $\{x \in s \mid K(x)\}$. (Note that s must not be free in K.) The iterator $x \in s'$ avoids the computation of K(x) for all elements x in s. Earley shows how to keep s' available in a program region R by means of incremental update rules, e.g., just before a slight change $s := s \cup \{a\}$ which adds the element a to s we can execute the following code,

if K(a) then $s' := s' \cup \{a\}$.

He also discusses setformer expressions

(2) $$\{x \in s \mid f(x) = q\}$$

whose value is the set of all elements x of the set s in which the value of the map f applied to the bound variable x equals the free variable q. To handle (2) efficiently, he constructs the map $T(q) = \{x \in s \mid f(x) = q\}$ defined over all values that q can have in a program region R containing (2). He indicates that the entire map T can be kept available in R whenever all changes to s in R are only element additions or deletions and all redefinitions of the map f change only one range value of f at a particular domain point. We will not describe his rules fully, since in the next chapter we intend to describe our own similar but more powerful transformations in a formal style.

In exploring a way to fit Earley's techniques into a

general framework, Schwartz noted that to reduce expressions such as (2), it is necessary to deal with parameters such as q upon which expressions do not depend continuously [Sch 7]. The present thesis develops the concept of formal differentiation further and applies it to a wider class of expressions under broader conditions than have been considered.

In rudimentary form our idea which extends the basic notions previously described (cf. p. 278) may be stated in the following way. We consider as reduction candidates those expressions

(3) $$C = f(x_1,\ldots,x_n)$$

in a strongly connected region R which are continuous with respect to redefinitions to some of their parameters $x_1,\ldots,x_k$ and discontinuous relative to changes to the others, $x_{k+1},\ldots,x_n$. Our formal differentiation method draws on the fact that if the final group of parameters is given constant values, then the expression $f' = f(x_1,\ldots,x_k,q_{k+1},\ldots,q_n)$ is continuous in its remaining parameters. To reduce (3) we can therefore use a 'memo' function $\bar{C}$ which keeps several values $\bar{C}(q_{k+1},\ldots,q_n) = f(x_1,\ldots,x_k,q_{k+1},\ldots,q_n)$ of f available in R. This makes it possible to avoid redundant calculations of (3) by replacing each such use by a retrieval operation $\bar{C}(x_{k+1},\ldots,x_n)$. The actual reduction transformation is sketched below:

i. On entrance to R initialize the mapping $\bar{C}$ by performing

$\bar{C}$:= *nullset*;

ii. Whenever any argument $x_1,\ldots,x_k$ is varied inside R, then the pre and post derivative operations necessary to keep each stored value $\bar{C}(q_{k+1},\ldots,q_n)$ available in R must be performed.

iii. When one of the variables $x_{k+1},\ldots,x_n$ changes inside R, no calculations need be made.

iv. Replace each calculation $f(x_1,\ldots,x_n)$ in R by code which either retrieves a stored value $\bar{C}(x_{k+1},\ldots,x_n)$ (if $[x_{k+1},\ldots,x_n]$ is in the domain of $\bar{C}$) or else calculates $f(x_1,\ldots,x_n)$ and records this value in $\bar{C}$. This code is roughly as follows:

```
(4)  if [x_{k+1},...,x_n] ∈ PROJECT(n-k,C̄)
          /* PROJECT returns the domain of C̄ */
       then C̄(x_{k+1},...,x_n)
       else C̄(x_{k+1},...,x_n) := f(x_1,...,x_n)
          /* use of an assignment side effect within
                          an expression */
```

The PROJECT function used in (4) is a projection operator on maps, and has the following simple SETL definition:

```
     DEFINEF PROJECT(m, MAP);
 /* if MAP is an n-ary SETL map viewed as a set of n+1-tuples,
    then when m ≤ n+1, PROJECT returns the set of m-tuples
    matching the first m component values of n+1-tuples in
    MAP */
     RETURN {x(1:m), x ∈ MAP};
     END;
```

The approach sketched above accepts the expense of map retrieval operations (iv) and incremental update code (ii) in order to eliminate redundant calculations of (3) in R. Note, however, that when the domain of $\bar{C}$ is large, then the cost of performing pre and post derivatives of f with respect to changes $x_j := \Delta_{x_j}$ (for $j \leq k$) may make formal differentiation unprofitable. For those fortunate cases in which the domain of $\bar{C}$ is small, or when we can guarantee that the derivative calculations can be limited to a small portion of the domain of $\bar{C}$, we will say that the discontinuity parameters $x_{k+1},\ldots,x_n$ of (3) are 'removable' and that the map $\bar{C}$ is continuous in the continuity parameters of f relative to the modifications occurring in R.

Chapter two develops the ideas outlined in the preceding discussion, and gives numerous examples and case studies of their application. Chapter three sketches an implementation design for a basic formal differentiation system which only handles expressions continuous in all of their parameters. Chapter four extends the initial design to one which can handle most of the examples presented in Chapter two that are not handled by the simpler system. Chapter four also considers various time and space improvements which could be incorporated in this extended design, and it concludes with some remarks proposing directions in future research.

II. BASIC CONCEPTS AND SET THEORETIC EXAMPLES

A. Preliminaries

Application of the idea of formal differentiation in a set-theoretic context was initiated by Earley and has been pushed further by Fong and Ullman [F1] who made the interesting observation that formal differentiation in a set-theoretic milieu could actually improve the asymptotic behavior of an algorithm and that this fact could be used to develop a theoretical characteriziation of the situations in which this technique applied. In the discussion which follows, we shall pursue Earley's idea in a less formal sense than that of Fong and Ullman, aiming to state pragmatic rules for the discovery and treatment of reasonably general cases in which formal differentiation can be applied. (Of course, any reasonable criterion for evaluating the utility of formal differentiation must rest on some notion of expected efficiency, albeit only a heuristic one. Such an informal complexity measure must at the very least distinguish between 'easy' and 'complicated' operators.)

In this chapter, we will be concerned principally with general sets and with mappings represented by sets of tuples. Unless otherwise specified, we assume a hash table implementation for sets in which entries are linked within a two way list, thus permitting a unit time membership test and a linear time search through sets. Moreover, if element addition and deletion can be performed directly on the body

of a set without copying, then these operations can also be done in unit time. We also assume a similar hashed implementation for maps in which various kinds of functional application and change are done in time proportional to a map's arity, and iteration through a map's domain takes linear time [AHU1, DGS1, P1, Schl]. In particular, the data structure for SETL maps described in DGS1 correspond to our assumptions.

B. Initial Examples

In SETL the computations s := s + {x} and s := s - {x} respectively add and delete the value of the element x from the set s. Both operations change s only 'slightly'. Similarly, if s and Δ are sets and the number of elements of Δ, $\#\Delta$, is much smaller than #s, then modifications of the form $s = s \pm \Delta$, represent 'slight' changes to s. We expect that such changes can be performed destructively at a cost proportional to $\#\Delta$ by the obvious technique which is written in SETL as follows:

```
(∀x ∈ Δ)                /* linear time search through Δ */
  s := s ± {x};;        /* destructive unit time assignment*/
```

If f is a set of pairs used as a SETL mapping, then the operation f(x) = z, which replaces all pairs whose first element is x by the pair [x,z] causes f to change slightly. If f is a set of (n+1)-tuples used as a multiparameter mapping, then the indexed assignment $f(x_1,\ldots,x_n) := z$ alters f only slightly.

The informal notion of 'slight' changes to a set can be used to illustrate the notion of expression 'continuity'. Examples of SETL expressions continuous in differential changes to all of their parameters are: set union s + t, set intersection, s * t, and set difference s - t.

Consider the set difference operation

(1) $$C = s - t .$$

If the value currently available for C is spoiled by a differential change $s := s \pm \Delta$ to s at a program point p, the value C of s - t can be restored on exit from p by executing the corresponding update code.

(2) $C := C + (\Delta - t)$, or $C := C - \Delta$

immediately prior to p. We will say concisely that the update code (2) gives a 'prederivative' of the expression (1) with respect to the change $s := s + \Delta$ or $s := s - \Delta$. Both redefinitions in (2) are slight changes to C and can usually be performed at less expense than the full calculation (1). Note that if (1) is performed in the obvious way, we can expect its running time to be proportional to #s, while the execution of each slight change in (2) will require time $\propto \#\Delta$ as realized by the following obvious implementation:

(3)
```
(∀x ∈ Δ)
  if x ∉ t then C := C ± {x}; ENDIF;
END∀;
```

Similarly, we see that (1) is continuous with respect to small changes t := t $\pm$ Δ. The prederivative code is simply

(4) C := C - Δ or C := C + (Δ * s) .

These are both slight modifications to C and can be performed in O(#Δ) expected steps.

Next consider the set union operation

(5) C = s + t

in which s and t can have overlapping values. The obvious implementation of this will run in O(#s + #t) time. The prederivatives of (5) with respect to changes s := s $\pm$ Δ are

(6) C := C + Δ or C := C - (Δ - t)

each of which will execute in time O(#Δ). Set intersection,

(7) C = s * t

has an obvious implementation requiring O(#s) steps. The prederivatives of (7) with respect to the slight changes s := s $\pm$ Δ are

(8) C := C + Δ * t or C := C - Δ

each of which has been seen to run in O(#Δ) steps.

If f is a 1-ary function, then the SETL range function (range of f on a set s), written as

(9) C = f[s]

is continuous with respect to s := s + Δ, and its prederivative code is:

(10) $C := C + f[\Delta]$.

The obvious implementation of (9) would take O(#s) elementary steps in contrast to the approximately O(#Δ) steps needed to perform (10). Note, however, that (9) is discontinuous with respect to s := s - Δ and with respect to indexed assignments such as f(y) := z .

The inverse image $f^{-1}[s]$ can be written in SETL using a set former expression

(11) $C = \{x \in DOM\ f \mid f(x) \in s\}$

where *DOM* f is the set of first components of pairs in f (if f has an arity of one then *DOM* f is the domain of f). Expression (11) is continuous in f, and for changes f(x) := y, its prederivative code is

(12) C := C - (if f(x) ∈ s then {x} else *nullset*)
+ (if y ∈ s then {x} else *nullset*).

The expected computational cost of an assignment (11) is O(#f). Clearly the prederivative (12) should execute in essentially constant time. Note, however that the C of (11) is discontinuous in s.

The operation f[s] may be continuous in s := s + Δ even if f is a programmed function. The conditional expression 'if a then s_1 else s_2' is continuous in s_1 and s_2 , but is, of course, discontinuous in its boolean parameter a.

As Earley has emphasized, expressions involving set-formers provide more interesting examples of this phenomenon of 'continuity'. The SETL expression

(13) $C = \{x \in s \mid f(x) = q\}$

which computes the set of all values of the set s such that the boolean valued subexpression f(x) = q holds, is a prototypical example. This expression is continuous in s and f, but discontinuous in q. If s is varied slightly by $s := s \pm \Delta$ then C can be updated by executing the prederivative

(14) $C := C \pm \{x \in \Delta \mid f(x) = q\}$

which represents a small change in C. When f is changed by executing the indexed assignment $f(y_0) := z$, then C can be updated by executing

(15) C := if $y_0 \in s$ then C - (if $f(y_0) = q$ then $\{y_0\}$ else *nullset*) + if z = q then $\{y_0\}$ else *nullset*;

just before the assignment $f(y_0) := z$.

More insight is gained by writing (15) as

(16) $C := C - \{x \in \{u \in s \mid u = y_0\} \mid f(x) = q\} + \{x \in \{u \in s \mid u = y_0\} \mid z = q\};$

since (16) begins to suggest a rule for updating more general set-theoretical expressions than (13). For example, before changing f by $f(y_0) := z$ the set

(17) $C_1 = \{x \in s \mid g(f(x)) = q\}$

can be updated by executing

(18) $C_1 := C_1 - \{x \in \{u \in s \mid u = y_0\} \mid g(f(x)) = q\}$
$+ \{x \in \{u \in s \mid u = y_0\} \mid g(z) = q\};$

Note, however that for (16) and (18) to be 'easy' calculations relative to (13) and (17) we require that each computation

of $\{u \in s \mid u = y_0\}$ occurring in this update code must be 'easy'. Of course, the automatic local transformation turning $\{u \in s \mid u = y_0\}$ into the equivalent and inexpensive operations

(19) *if* $y_0 \in s$ *then* $\{y_0\}$ *else nullset*

applies to the examples above.

The setformer

(20) $C_2 = \{x \in s \mid f(g(x)) = q\}$

which can be updated by executing

(21) $C_2 := C_2 - \{x \in \{u \in s \mid g(u) = y_0\} \mid f(g(x)) = q\}$
$+ \{x \in \{u \in s \mid g(u) = y_0\} \mid z = q\};$

can be handled by a combination of the transformations already mentioned. For C_2 to be continuous in s and f (relative to modifications $s := s \pm \Delta$ and $f(y_0) := z$ occurring in a strongly connected region R) all prederivative code and all other attendant code necessary to maintain available expressions on which the prederivatives depend must consist of 'easy' calculations relative to C_2. Since by (13) and (14) we know that when g and y_0 are invariant in R, the costly subexpression $\{u \in s \mid g(u) = y_0\}$ of (21) can be made available throughout R by inexpensive operations, C_2 is seen to be continuous in its parameters s and f. All the updating operations (16), (18) and (21) are to be performed just *prior* to the change $f(y_0) := z$ for which they compensate.

Next consider the case of a set-theoretic expression in whose defining condition f appears twice with different arguments, as for example

(22) $C_3 = \{x \in s \mid f(g(x)) = f(h(x))\}$

or

(23) $C_4 = \{x \in s \mid f(f(x)) = q\}$.

Before changing f by $f(y_0) := z$, we can update such sets by computing a set s_0 of all those elements of s over which the boolean condition appearing in the setformer (22) or (23) can change value as a result of the indexed assignment to f. Then after the change to f we can adjust the value of the setformer for those points of s_0. In the case of C_3 and C_4 this leads us to the following prederivative and postderivative updating operations:

(24)
```
s0 := {x ∈ s | g(x) = y0 or h(x) = y0};
f(y0) := z;
(∀x ∈ s0)
   if f(g(x)) = f(h(x)) then
      C3 := C3 + {x}; else
      C3 := C3 - {x};
   endif;
end ∀;
```

and

(25)
```
s0 := {x ∈ s | f(x) = y0 or x = y0}
f(y0) := z;
(∀x ∈ s0)
```

```
    if f(f(x)) = q    then
        C4 := C4 + {x}; else
        C4 := C4 - {x};
    endif;
end ∀;
```

In both (24) and (25) each setformer s_0 must be continuous in its parameters for C_3 and C_4 to be continuous. If, for example, (23) occurs in a strongly connected region R and within R all redefinitions to s and f are slight, if the free variable q of (23) is invariant and each parameter y of an indexed assignment f(y) := z appearing in R is also invariant, then the auxiliary set s_0 can be made available within R at small cost. Moreover, although s_0 depends continuously on f, it is interesting to note that s_0 is not spoiled by $f(y_0)$:= z in either of the code sequences (24) or (25); this observation facilitates our method of keeping s_0 available in R.

C. Formal Differentiation of Set Theoretical Expressions Continuous in All of Their Parameters

We now formulate a few general rules concerning the formal differentiation of set theoretical expressions continuous in all of their free parameters. It must be observed that none of the transformations which we are studying can safely be applied to expressions containing operations which cause side effects for which reason we shall

always assume such operations to be absent in the expressions we treat. We also assume that typefinding is applied prior to any attempt to optimize by formal differentiation so that object types are known during the analysis of a program for reduction (cf.,(T1) for a method of type analysis for SETL).

Consider the set-theoretic expression

(1) $$C = \{x \in s \mid K(x)\}$$

in which K(x) is any boolean-valued subexpression containing only free occurrences of the bound variable x, and containing no free instance of the set, s. Recall that a full calculation of (1) as performed by the following standard procedure for set formers,

(2)
```
C := nullset;
(∀x ∈ s)                    / linear time search /
  if K(x) then C:=C+{x}; / execute K(x) /
  endif;
end ∀;
```

can take O(#s × cost(K(x))) steps. An expression (1) is continuous with respect to slight changes s := s $\pm$ Δ to s since the prederivatives of (1) with respect to these changes are also 'easy' calculations,

(3) $$C := C \pm \{x \in \Delta \mid K(x)\} .$$

Suppose that the expression (1) is used in a strongly connected region R and that the following conditions hold:

(i) Inside R, s is only changed by slight modifications of the form s := s $\pm$ Δ, where Δ is a small set in

comparison with s.

(ii) The set valued variable s is defined on entrance to R.

(iii) Aside from s all other parameters on which (1) depends are region constants of R.

Then we can formally differentiate the expression (1) in R. If all these conditions apply, then formal differentiation of (1) is accomplished by applying the following rule.

Rule 1. We begin by making (1) available on entrance to R. This is done by inserting the assignment $C := \{x \in s \mid K(x)\}$ into R's initialization block. Then, at each point p inside R where the value of s changes by $s := s \pm \Delta$, the value of C (which could be spoiled at p) is updated by inserting the prederivative code (3). All calculations (1) are redundant in R and can be replaced by uses of the variable C.

We remark here that the rule just described is complex enough to illustrate the difficulties bound to be encountered in any serious effort to automatically guarantee improvement in running time by formal differentiation. Several pragmatic and simplifying assumptions are implicit in our expectation that formal differentiation of (1) is worthwhile:

(i) We only consider the cost of destructive assignments to the set C as when C undergoes slight modifications within (2)

and (3); we ignore the costs of making fresh copies of C on occasions when only nondestructive assignments are possible.

(ii) We pay no attention to the cost of rehashing elements of C when C grows too large.

(iii) We do not figure the cost of increased garbage collection which might be required after formal differentiation, since extra space is required to store the values of available expressions possibly over large program regions.

If these assumptions fail, the actual effect on a program's running time of applying Rule 1 may be undesirable. (This effect depends on such facts as frequency information and relative sizes of sets, undecidable at compile time.) Moreover, to replace these assumptions and restrictions by others may lock us into an unrealistic model of limited utility. (For further discussion of this point refer to Appendix B.)

Rule 1 can be used to derive another more comprehensive rule for formal differentiation of expressions like (1).

Suppose that the boolean subexpression K of (1) contains m free occurrences of the n-ary mapping symbol f. Suppose also that these m occurrences of f appear in r different terms,

$$f(p_{11}(x),\ldots,p_{1n}(x)),f(p_{21}(x),\ldots,p_{2n}(x)),\ldots f(p_{r1}(x)\ldots p_{rn}(x))$$

where $p_{ij}(x)$ represents the j-th parameter expression (involving x which is the bound variable of the set former) of the i-th term. Then as derivatives of (1) with respect

to indexed assignments $f(y_1,\ldots,y_n) := z$ we can use either of the following code sequences

	Relative Position	Derivative Code
(5)	p-2	$s_0 := \{x \in s \mid p_{11}(x)=y_1 \& \ldots \& p_{1n}(x) = y_n$ *or ... or* $p_{r1}(x) = y_1 \& \ldots \& p_{rn}(y) = y_n\}$;
	p-1	$C := C - \{x \in s_0 \mid K(x)\}$;
	p	$f(y_1,\ldots,y_n) := z$;
	p+1	$C := C + \{x \in s_0 \mid K(x)\}$;

or

(5')	p-1	$s_0 := \{x \in s \mid p_{11}(x)=y_1 \& \ldots \& p_{1n}(x) = y_n$ *or ... or* $p_{r1}(x) = y_1 \& \ldots \& p_{rn}(x) = y_n\}$;
	p	$f(y_1,\ldots,y_n) := z$;
	p+1	$(\forall x \in s_0)$ *if* $K(x)$ *then* $C := C+\{x\}$; *else* $C := C-\{x\}$; *endif*; *end* $\forall$;

It is not difficult to see that (5') has the same effect as (5). Moreover, it can be shown that the validity of (5) is a corollary of rule 1. To see this, consider the set $D_{f_i} = \{[p_{i1}(x)....,p_{in}(x)] \mid x \in s\}$. Let p_i be the mapping whose domain is s and where $p_i(x)=[p_{i1}(x),\ldots,p_{in}(x)]$.

Then for any n-tuple $[y_1,\ldots,y_n]$, we have $p_i^{-1}(y_1,\ldots,y_n) = \{x \in s \mid p_{i1}(x) = y_1 \;\&\ldots\&\; p_{in}(x) = y_n\}$. If s changes by deletion of $p_i^{-1}(y_1,\ldots,y_n)$, then D_{f_i} changes by deletion of the n-tuple $[y_1,\ldots,y_n]$. Moreover if s is modified by deletion of $\bigcup_{i=1}^{r} p_i^{-1}(y_1,\ldots,y_n)$, then the n-tuple $[y_1,\ldots,y_n]$ is removed from the domain of all the f terms occurring in (1). Next we observe that if C is available on entrance to p (i.e., is available just prior to the modification to f by the indexed assignment $f(y_1,\ldots,y_n) := z$), and if $[y_1,\ldots,y_n] \notin \bigcup_{i=1}^{r} D_{f_i}$ just before point p, then the statement $f(y_1,\ldots,y_n) := z$ does not change any of the occurrences of f in (1). Consequently, C is not spoiled by the indexed assignment, and it remains available.

Suppose now that in expression (1) C is available on entrance to the program point p. Then we would proceed as follows: (1) at p-3, put s_0 equal to the set $\bigcup_{i=1}^{r} p_i^{-1}(y_1,\ldots,y_n)$, (2) at p-1 delete s_0 from s; (3) at p-2 update C in accordance with rule 1, (4) at p+2 add s_0 back to s; and (5) at p+1, use rule 1 again to update C. This would give us the following code:

$$
\begin{array}{lll}
 & p\text{-}3 & s_0 := \{x \in s \mid p_{11}(x)=y_1 \,\&\, \ldots \,\&\, p_{in}(x) = y_n \\
 & & \qquad\qquad \textit{or} \ \ldots \ \textit{or} \\
 & & \qquad p_{r1}(x) = y_1 \,\&\, \ldots \,\&\, p_{rn}(x) = y_n\} \\
 & p\text{-}2 & C := C - \{x \in s_0 \mid K(x)\} \\
(6) & p\text{-}1 & s := s - s_0 \\
 & p & f(y_1,\ldots,y_n) := z \\
 & p\text{+}1 & C := C + \{x \in s_0 \mid K(x)\} \\
 & p\text{+}2 & s := s + s_0
\end{array}
$$

In this code C is not spoiled by the statement $f(y_1,\ldots,y_n) := z$. Hence, if C is available upon entrance to p-3, then by rule 1 we know that C remains available on exit from p+2. And now finally, since in (6) the value of the set s is the same before p-1 as after p+2, and because s is not used between p-1 and p+2, the code (6) is equivalent to that shown in (5). The assumption that at least one of the parameters in each f term in K involves x (the bound variable of the set former) will usually cause the set s_0 to be small in comparison with s.

To show that (1) is continuous relative to differential modifications of the form $s := s \pm \Delta$ (where Δ is small in comparison with s) and also relative to indexed assignments $f(y_1,\ldots,y_n) := z$ occurring in a strongly connected region R, we want to insure that the pre and post derivative code in (5)

and (5') can be made to consist of 'easy' calculations. For this to be the case, the setformer

$$(7)\quad s_0 = \{x \in s \mid p_{11}(x) = y_1 \;\&\ldots\&\; p_{1n}(x) = y_n \quad \textit{or} \ldots \textit{or} \quad p_{r1}(x) = y_1 \;\&\ldots\&\; p_{rn}(x) = y_n\}$$

occurring at points p-2 of (5) and p-1 of (5') must be profitably reducible; and this will be true if (7) is continuous in s and f. By Rule 1 we know that (7) is continuous in s. We shall see that (7) is also continuous with respect to indexed assignments $f(w_1,\ldots,w_n) := v$ occurring in R if for every such assignment each parameter expression $w_1,\ldots,w_n$ is a region constant of R.

Before exhibiting actual inexpensive update code supporting this claim, it is useful to look at the situation from a somewhat different point of view. Let d be the maximum depth of nesting of f terms contained within other f terms in the boolean subpart K of (1) (e.g., the depth of the term $f(g(f(x + f(0))))$ is 3). Then (7) has a maximum nesting depth of d-1. We will show inductively that for $d = 1,2,\ldots$ (7) is continuous in f. For d = 1 (7) contains no f terms and is trivially continuous. If d = 2 is the nesting of (1) then the depth of (7) is 1, and (7) can be reduced economically using either rules (5) or (5') and Rule 1. Next assume that any expression (7) which has depth d-1 less than or equal to k-1 is continuous in f, and consider the case d = k + 1. Since the depth of (7) is d,

application of rules (5) or (5') to (7) will produce an expression s_0' which is of the same form as (7) but with a depth d-1. By hypothesis, s_0' is continuous in f. Thus, we can conclude that (1) is continuous in f and s for any depth d.

The previous remarks suggest that the derivative of (1) with respect to f should be realized by first applying rules (5) or (5') to (1); this gives rise to an expression s_1 of the same form as (7). Then either (5) or (5') can be applied to s_1 and to each successive s_j , j = 2,...,d-1 emerging by use of (5) or (5') until the d-1'st derivative is applied. The final expression s_{d-1} produced by this process will have zero depth and can be reduced by Rule 1.

This approach will often be feasible, but in general it is not easy to say whether the d different auxiliary sets which must be kept available in R as a result of the transformations sketched above will overlap strongly and, hence, require excessive space. Nor in general can we say how much of an improvement in speed (if any) is gained by maintaining all these sets in addition to (1) by incremental calculations.

Fortunately we can suggest a much more attractive transformation which introduces fewer auxiliary calculations. Suppose that the expression (1) is used in a strongly connected region R and that the following conditions hold:

(i) The boolean valued subexpression K(x) contains m free occurrences of an n-ary mapping f (in which each such occurrence has at least 1 parameter expression involving x,

the bound variable of the set former); all other free variables occurring in K are loop invariant.

(ii) The m occurrences of f in K begin r distinguishable f terms, $f(p_{11}(x),\ldots,p_{1n}(x)),\ldots,f(p_{r1}(x),\ldots,p_{rn}(x))$.

(iii) Inside R, s is only changed by slight modifications of the form $s := s \pm \Delta$, where Δ is a small set in comparison with s, and f is only changed by indexed assignments of the form

$$f(y_1,y_2,\ldots,y_n) := z \ . \tag{8}$$

Then we can formally differentiate the expression (1) in the region R. The differentiation rule is as follows:

Rule 2. (There are two cases to consider.)

Case 1. Consider the class of expressions (1) in which for $i = 1,\ldots,r$ and $j = 1,\ldots,n$ each parameter expression $p_{ij}(x)$ either does not involve x or does involve x and can be symbolically transformed into a linear factor of the forms $x*a\pm b$. For any expression (1) in this class (e.g., expressions (13) and (17) of section B), we can use an inexpensive variant of (5) to update (1) with respect to indexed assignments (8), even when the parameters $y_1,\ldots,y_n$ appearing in (8) are not region constants. This variant of (5) is obtained if we simplify the setformer (7) used at point p-2 of (5) into the following efficient form,

(9) if $q_1 \in s \ \&_{j=1}^{n-1} \ t_{1j} = t'_{1j}$ then $\{q_1\}$ *else nullset*

+... +

if $q_r \in s \ \&_{j=1}^{n-1} \ t_{rj} = t'_{rj}$ then $\{q_r\}$ *else nullset.*

in which q_i , t_{ij} , and t'_{ij} for i = 1,...,r and j = 1,...,n are meta symbols denoting computed expressions.

We derive (9) from (7) by the following straightforward manipulation. Expand (7) into a union of setformers $\{x \in s \mid \&_{j=1}^{n} (p_{1j}(x) = y_j)\} + \ldots + \{x \in s \mid \&_{j=1}^{n} (p_{rj}(x) = y_j)\}$. By assumption we know that within each set former, $\{x \in s \mid \&_{j=1}^{n} (p_{ij}(x) = y_j)\}$, i = 1,...,r there must be a parameter expression, say $p_{i1}(x)$, which involves x. Hence, we can transform the equality $p_{i1}(x) = y_1$ into the form $x = q_i$ and rewrite each conjunction $\&_{j=1}^{n} (p_{ij}(x) = y_j)$ as $x = q_i \ \&_{j=2}^{n} (p_{ij}(q_i) = y_j)$ which involves only one occurrence of x. If we now make the substitution t_{ij-1} for $p_{ij}(q_i)$ and t'_{ij-1} for y_j , i = 1,...,r, j = 2,...,n, each resulting setformer $\{x \in s \mid x = q_i \ \&_{j=1}^{n-1} (t_{ij} = t'_{ij})\}$ may be simplified to the following conditional expression,

if $q_i \in s \ \&_{j=1}^{n-1} (t_{ij} = t'_{ij})$ then $\{q_i\}$ *else nullset.*

Case 2. To reduce more general expressions whose depth of nesting in f terms is greater than 1 (e.g., (20), (22), and (23) of the last section), we must use a different method. We will at first consider only situations in which each parameter $y_1,\ldots,y_n$ of indexed assignments $f(y_1,\ldots,y_n) := z$ occurring in R is a region constant of R. In the next section, however, this restriction will be cast off.

Suppose that R contains t different indexed assignments to f which we denote by $f(y_{11},\ldots,y_{n1}) := z_1,\ldots,$ $f(y_{1t},\ldots,y_{nt}) := z_t$. Then on entrance to R, we must insert the following initializing code,

$$
\begin{aligned}
(10)\ C &:= \{x \in s \mid K(x)\}; && \text{/* expression (1) */}\\
s_0^{(1)} &:= \{x \in s \mid \mathop{or}_{i=1}^{r}\ (\mathop{\&}_{j=1}^{n}\ p_{ij}(x) = y_{j1})\}; && \text{/* set former (7)*/}\\
s_0^{(2)} &:= \{x \in s \mid \mathop{or}_{i=1}^{r}\ (\mathop{\&}_{j=1}^{n}\ p_{ij}(x) = y_{j2})\};\\
&\ \ \vdots\\
s_0^{(t)} &:= \{x \in s \mid \mathop{or}_{i=1}^{r}\ (\mathop{\&}_{j=1}^{n}\ p_{ij}(x) = y_{jt})\}; && \text{/* based on */}\\
& && \text{/* } f(y_{1t},\ldots y_{nt}) := z_t \text{ */}
\end{aligned}
$$

Whenever s is modified in R, we then apply Rule 1 to update C, $s_0^{(1)}, s_0^{(2)}, \ldots, s_0^{(t)}$. At each program point in R at which f is changed, we keep C, $s_0^{(1)}, s_0^{(2)}, \ldots, s_0^{(t)}$ available by executing either of the following code sequences, which are based on (5) and (5').

$$
\begin{aligned}
(11)\ s_0^{(1)} &:= s_0^{(1)} - \{x \in s_0^{(\ell)} \mid \mathop{or}_{i=1}^{r}\ (\mathop{\&}_{j=1}^{n}\ p_{ij}(x) = y_{j1})\};\\
&\ \ \vdots\\
s_0^{(t)} &:= s_0^{(t)} - \{x \in s_0^{(\ell)} \mid \mathop{or}_{i=1}^{r}\ (\mathop{\&}_{j=1}^{n}\ p_{ij}(x) = y_{jt})\};\\
C &:= C - \{x \in s_0^{(\ell)} \mid K(x)\};\\
& f(y_{1\ell},\ldots,y_{n\ell}) := z\ ; \quad \text{/* all } s_0 \text{ sets are updated}\\
& \qquad\qquad \text{except } s_0^{(\ell)} \text{ */}\\
C &:= C + \{x \in s_0^{(\ell)} \mid K(x)\};\\
s_0^{(1)} &:= s_0^{(1)} + \{x \in s_0^{(\ell)} \mid \mathop{or}_{i=1}^{r}\ (\mathop{\&}_{j=1}^{n}\ p_{ij}(x) = y_{j1})\};\\
&\ \ \vdots\\
s_0^{(t)} &:= s_0^{(t)} + \{x \in s_0^{(\ell)} \mid \mathop{or}_{i=1}^{r}((\mathop{\&}_{j=1}^{n}\ p_{ij}(x) = y_{jt})\};
\end{aligned}
$$

or

(11') $f(y_{1\ell},\ldots,y_{n\ell}) := z_\ell$;

($\forall x \in s_0^{(\ell)}$)

if K(x) *then* C := C + {x}; *else* C := C-{x}; *endif*;

if $\overset{r}{\underset{i=1}{or}}$ ($\overset{n}{\underset{j=1}{\&}}$ $p_{ij}(x) = y_{jl}$) *then* $s_0^{(1)} := s_0^{(1)}-\{x\}$;

else $s_0^{(1)} := s_0^{(1)}+\{x\}$; *endif*;

: /* all sets s_0 except $s_0^{(\ell)}$ are updated */

if $\overset{r}{\underset{i=1}{or}}$ ($\overset{n}{\underset{j=1}{\&}}$ $p_{ij}(x) = y_{jt}$) *then* $s_0^{(t)} := s_0^{(t)}-\{x\}$;

else $s_0^{(t)} := s_0^{(t)}+\{x\}$;*endif*;

end $\forall$;

This keeps C available throughout R, so that all calculations (1) can be replaced by uses of C.

It is easy to see that (11') computes the same thing as (11). To justify the code sequence (11), we need to substantiate two claims: (1) $s_0^{(\ell)}$ cannot be spoiled by assignments $f(y_{1\ell},\ldots,y_{n\ell}) := z_\ell$; and (2) for any m, $1 \leq m \leq t$ and $m \neq \ell$, the derivative code shown in (11) for $s_0^{(m)}$ is correct.

To justify claim (1), we consider the predicates $Q_i(x) = \overset{n}{\underset{j=1}{\&}}$ $p_{ij}(x) = y_{j\ell}$, i = 1,...,r. Let w be an element of $s_0^{(\ell)}$ just prior to the program point p at which $f(y_{1\ell},\ldots,y_{n\ell}) := z_\ell$ occurs. Then for some k, $1 \leq k \leq r$, $Q_k(w)$ must hold before p. Moreover, we can choose k in such a way that among those predicates $Q_1(w),\ldots,Q_r(w)$ that hold before p, $Q_k(w)$ has a minimal depth d of nesting

in f terms. Consequently, $Q_k(w)$ must hold after p and w must be an element of $s_0^{(\ell)}$ after p. For otherwise, an f term occurring in $Q_k(w)$ would be spoiled at p. And this implies that a predicate $Q_v(w)$, $v \neq k$, with a smaller nesting depth than $Q_k(w)$ would hold just before p -- a contradiction.

In proving claim (2) it is useful to organize the r different f terms occurring in (1) as follows: Let $f(p_{k1}(x),\ldots,p_{kn}(x))$, $k = 1,\ldots,q$ be all those f terms which never occur within any of the f terms of(1). (These are all the f terms which also have no occurrences in $s_0^{(m)}$, $m = 1,\ldots,t$.) Then application of the transformation (5) to update $s_0^{(m)}$, $1 \leq m \leq t$ and $m \neq \ell$ in a manner compensating for the change $f(y_{1\ell},\ldots,y_{n\ell}) := z$ to f results in the following code:

$$s_0 := \{x \in s \mid \underset{i=q+1}{\overset{r}{or}} \left(\underset{j=1}{\overset{n}{\&}} p_{ij}(x) = y_j \right)\};$$

$$s_0^{(m)} := s_0^{(m)} - \{x \in s_0 \mid \underset{i=1}{\overset{r}{or}} \left(\underset{j=1}{\overset{n}{\&}} p_{ij}(x) = y_{jm} \right)\};$$

$$f(y_{1\ell},\ldots,y_{n\ell}) := z_\ell;$$

$$s_0^{(m)} := s_0^{(m)} + \{x \in s_0 \mid \underset{i=1}{\overset{r}{or}} \left(\underset{j=1}{\overset{n}{\&}} p_{ij}(x) = y_{jm} \right)\}.$$

But since $s_0 \subseteq s_0^{(\ell)}$, it follows from the proof of (5) (cf. discussion of (6)) that we can replace occurrences of s_0 by $s_0^{(\ell)}$ in the code just above.

The code generated by rule 2 can be improved by eliminating redundancies in the expression $\{x \in s_0 | K(x)\}$ which appears at locations p-1 and p+1 of (5). Suppose we know that $s_0 = \bigcup_{i=1}^{r} R_i$, where $R_1,\ldots,R_r$ are disjoint sets. Then $\{x \in s_0 | K(x)\}$ can be rewritten as $\bigcup_{i=1}^{r} \{x \in R_i | K(x)\}$. Suppose also that in each set $\{x \in R_i | K(x)\}$, K(x) can be transformed (by elimination of redundant operations) into an equivalent but easier to evaluate expression $K_i(x)$. Then it may be worthwhile to work with the partition $\{R_i\}$ of s_0 instead of s_0 and to rewrite

$$\{x \in s_0 | K(x)\} \quad \text{as} \quad \bigcup_{i=1}^{r} \{x \in R_i | K_i(x)\}.$$

As an example of this, observe that if we let $R_i = \{x \in (s - \bigcup_{k=0}^{i-1} R_k) | P_{i1}(x) = y_1 \ \&\ldots\& \ P_{in}(x) = y_n\}$, where $R_0 = \emptyset$, then $R_1,\ldots,R_r$ form a partition of s_0. Moreover, on the set R_i we can replace the term $f(p_{i1}(x),\ldots,p_{in}(x))$ which appears in the expression K at location p-1 of the code generated by rule 2 by $f(y_1,\ldots,y_n)$ (cf. (5) above). This can lead to a version of line p-1 of (5) which is relatively easy to evaluate, and it is therefore tempting to apply the same transformation to line p+1 of (5). However, at location p+1 we cannot, even after breaking up $\{x \in s_0 | K(x)\}$ into $\bigcup_{i=1}^{r} \{x \in R_i | K(x)\}$, simply replace each term $f(p_{i1}(x),\ldots,p_{in}(x))$ in K by z. This is because the indexed assignment appearing in line p of (5) changes f and may therefore cause some parameter $p_{ij}(x)$

appearing in $\{x \in R_i | K(x)\}$ within (5) and containing an occurrence of f to have a value different from y_j. When dealing with cases complicated enough for this problem to arise, we can make use of a second, finer, partition $R'_1,\ldots,R'_r$ of s_0 defined as follows: First set $R'_0 := \emptyset$ as before. Next find all f terms $f_{i_1},\ldots,f_{i_{r_0}}$ whose parameter expressions involve no f term, and put

$$R'_1 := \{x \in s \mid p_{i_1 1}(x) = y_1 \,\&\ldots\&\, p_{i_1 n}(x) = y_n\} ,$$

$$R'_2 := \{x \in (s-R'_1) | p_{i_2 1}(x) = y_1 \,\&\ldots\&\, p_{i_2 n}(x) = y_n\} , \ldots,$$

$$R'_{r_0} := \{x \in (s-\bigcup_{k=0}^{r_0-1} R'_k) | p_{i_{r_0} 1}(x) = y_1 \,\&\ldots\&\, p_{i_{r_0} n}(x) = y_n\}.$$

After this, find all f terms $f_{i_{r_0+1}}$, $f_{i_{r_0+2}}$,...,$f_{i_{r_1}}$ which do not belong to the set $F_1 = \{f_{i_1},\ldots,f_{i_{r_0}}\}$ but whose parameter expressions only contain f terms which do belong to F_1. Define sets $R'_{r_0+1},\ldots,R'_{r_1}$ by writing

$$R'_\ell := \{x \in (s-\bigcup_{k=0}^{\ell-1} R'_k) | p_{i_\ell 1}(x) = y_1 \,\&\ldots\&\, p_{i_\ell n}(x) = y_n\}.$$

Iterating this procedure sufficiently often we will obtain a partition $\{R'_1,\ldots,R'_r\}$ which can be used to eliminate redundant calculations of $f(p_{i1}(x),\ldots,p_{in}(x))$ at both p-1 and p+1. More specifically, if we let $K(x)_{s_1,\ldots,s_n}[t_1,\ldots t_n]$ denote the result of substituting the terms $t_1,\ldots,t_n$ for the terms $s_1,\ldots,s_n$ occurring in K(x), we can replace the code occurring at location p-1 (in rule 2) by

$$C := C - \bigcup_{i=1}^{r} \{x \in R_i' | K(x)_{p_{i1}(x),\ldots,p_{in}(x)}[y_1,\ldots,y_n]\}$$

and the code occurring at p+1 by

$$C := C + \bigcup_{i=1}^{r} \{x \in R_i' | (K(x)_{f(p_{i1}(x),\ldots,p_{in}(x))}[Z]_{p_{i1}(x),\ldots p_{in}(x)}[y_1,\ldots,y_n]\}.$$

(Note that the immediately preceding formula describes two successive steps of substitution.)

This general method allows the code used to reduce various set former expressions in examples (20)-(25) of Section B above to be generated automatically.

As an example of the redundancy elimination method just outlined, consider the following expression

(12) $C = \{x \in s | f(f(f(x+1) + 1)) = f(f(x+1) + 1)\}$.

Suppose that the mapping f is changed slightly by an indexed assignment, $f(y_0) = Z$ which occurs at a program point p. Then to update the value of (12) we proceed as follows. First a partition R_1, R_2, R_3 is computed. Observe that this partition contains three sets because only three different f terms occur in the boolean subexpression in (12): these are f(x + 1), f(f(x+1) + 1), and f(f(f(x+1) + 1)). Since f(x+1) is the only f term of (12) whose parameter expression involves no f term, we put $R_1 := \{x \in s | (x+1) = y_0\}$. Since the parameter part of f(f(x+1) + 1) involves f(x+1),

we set

$$R_2 := \{x \in (s\text{-}R_1) \mid f(x+1) + 1 = y_0\} \text{ and}$$

$$R_3 = \{x \in (s\text{-}(R_1+R_2)) \mid f(f(x+1) + 1) = y_0\}.$$

The code generated to update (12) is then as follows:

(13) $R_1 := \{x \in s \mid x + 1 = y_0\};$

$R_2 := \{x \in (s\text{-}R_1) \mid f(x+1) + 1 = y_0\};$

$R_3 := \{x \in (s\text{-}(R_1+R_2)) \mid f(f(x+1) + 1) = y_0\};$

$C := C - \{x \in R_1 \mid f(f(f(y_0) + 1)) = f(f(y_0) + 1)\}$

$\quad - \{x \in R_2 \mid f(f(y_0)) = f(y_0)\}$

$\quad - \{x \in R_3 \mid f(y_0) = y_0\};$

$f(y_0) := Z;$

$C := C + \{x \in R_1 \mid f(f(Z+1)) = f(Z+1)\}$

$\quad + \{x \in R_2 \mid f(Z) = Z\} + \{x \in R_3 \mid Z = y_0\};$

We note that the set former expressions defining R_1, R_2 and R_3 in (13) are continuous in all parameters with the exception of y_0 (which we temporarily require to be a region constant), so that they can be made available in R economically using techniques already described.

As noted by Earley, the method of formal differentiation which has been described can be extended in a useful way to apply to various SETL expressions that implicitly contain set formers. Among these are the forall iterator

(i.e., (∀x ∈ s|K(x)) *block*), the existential and universal quantifiers (i.e., ∃x ∈ s|K(x) and ∀x ∈ s|K(x)), and the compound operator (i.e.,

[<binop>: x ∈ s | K(x)] e(x)).

To formally differentiate these expressions, we rewrite them by replacing the implicit set former subpart, x ∈ s|K(x), which they contain with x ∈ {u ∈ s|K(u)}. The set former subexpressions thus exposed can then be differentiated using rules 1 and 2.

Let us now consider more closely the SETL compound operation

(14) C_1 = [*binop*: x ∈ C] e(x) ,

an illustrative example of which [+: x ∈ C] e(x) calculates the value $\sum_{x \in C} e(x)$. In general, [*binop*: x ∈ C] e(x) means $e(x_1)$ *binop*...*binop* $e(x_n)$ where C = $\{x_1,\ldots,x_n\}$. For the general case in which the binary operation *binop* has an appropriate inverse, *inverse binop* (e.g. arithmetic binary + with - as its inverse), we note that (14) is continuous relative to slight changes in C; i.e., before an occurrence of the code C := C $\pm$ Δ, C_1 can be updated by an appropriate inexpensive change, either

(15) C_1 := C_1 *binop*[*binop*: x ∈ (Δ - C)] e(x);

or

(15') C_1 := C_1 *inverse binop* [*binop*: x (Δ*C)] e(x);

Applying the heuristic rule 'continuous functions of continuous functions are continuous' to C_1 of (15) and C of (1) yields update identities for a more general compound operation form

(16) C = [*binop*: x ∈ s | K(x)] e(x) .

In order to formally differentiate the expression (16) in a strongly connected region R, we require all the conditions imposed on (1) to hold, and also require that neither the set s nor the n-ary mapping symbol f occurring in K should appear in the subexpression e of (16). If all these conditions are met, we differentiate (16) by first making it available on entrance to R. This is accomplished by inserting the assignment (16) into R's initialization block. Next, within R at each point p where C can be spoiled by 'slight' modifications to the variables s or f, we can apply the following continuity rules for (16) that parallel rules 1 and 2:

Rule 3: where s is modified in R by the code $s = s \pm \Delta$, the value of (16) can be maintained in C by executing

(17) C := C *binop*[*binop*: x ∈ (Δ - s) | K(x)] e(x)

or

(17') C := C *inverse binop*[*binop*: x ∈ Δ * s|K(x)] e(x)

respectively.

A similar rule analogous to rule 2 can be stated to cover the case of changes to f.

Rule 4. Suppose that the Boolean expression K(x) of (1) contains m free occurrences of the n-ary mapping f. Use the same notation and enabling conditions explained in connection with Rule 2. Then at each point p in the region R at which f is changed by an indexed assignment, the following code transformation should also be made:

Relative Position	Derivative Code
p-2	s_0 := {x∈s|$p_{11}(x) = y_1$ &...& $p_{1n}(x) = y_n$ *or ... or* $p_{r1}(x) = y_1$ &...& $p_{rn}(x) = y_n$};
p-1	C := C *inverse binop* [*binop*:x ∈(s_0*s) | K(x)] e(x)
p	$f(y_1,...,y_n)$:= Z;
p+1	C := C<binop>[<binop>: x∈(s_0-s)|K(x)]e(x)

It is easily seen that Rule 4 follows from Rule 3 in much the same way that Rule 2 follows from Rule 1.

These rules imply continuity properties for many other high level SETL operations. The counting operation applied to a set former; i.e., #{x ∈ s | K(x)} can be treated as [+: x ∈ s | K(x)] 1. When side effects of the existential and universal quantifiers can be ignored, then the corresponding SETL forms ∃x ∈ s | K(x) and ∀x ∈ s | K(x) can be rewritten as [+: x ∈ s | K(x)] 1 ¬= 0 and [+: x ∈ s | ¬K(x)] 1 = 0 respectively. Set inclusion (the predicate R ⊇ S) is continuous in both S and R since in SETL, R *incs* S can be handled as [+: x ∈ S | x ∉ R] 1 = 0.

Although iterative operations formed using range iterators, e.g., LO $\leq$ n < HI|K(n) can be differentiated using the techniques already mentioned, range iterators are frequently amenable to other reduction methods discussed by Schwartz [Sch8]. Consider as an example the following existential quantifier,

(18) $\exists$ LO $\leq$ n < HI | K(n)

where K does not depend on the integer valued variables LO or HI. In this case, rather than differentiating the full expression (18) we can reduce one or both range boundaries so as to limit the size of the search which implements (18).

In particular, if we want to reduce the lower boundary of (18), we can rewrite (18) as

(19) [*min*: LO $\leq$ n < HI | K(n)]n $\neg=$ Ω

allowing the compound *min* operation to be differentiated. The prederivative of C = [min: LO $\leq$ n < HI|K(n)]n with respect to LO := LO - Δ is described by the following update code,

```
(20)  if Δ ≥ 0 then
          IF ∃ LO - Δ ≤ n < LO|K(n) then C := n;
          ENDIF;
        else C := [min: LO-Δ ≤ n < HI|K(n)]n;
      ENDIF;
```

If an n-ary map f occurs in r distinguishable terms of K and each such term depends on n, we can apply a rule analogous to Rule 2 to differentiate C with respect to indexed assignments, $f(y_1,\ldots,y_n) := z$. The update code is

(21)
```
T := [LO ≤ n < C| or(i=1..r) ( &(j=1..n) p_ij(n) = y_j)];
f(y_1,...,y_n) := Z;
if ∃n ∈ T | K(n) then C := n;
else if ¬K(C) then C := [min: C+1 ≤ n < HI|K(n)]n;
ENDIF;
```

It is not difficult to see that the technique described above can also be applied to other SETL operations such as universal quantifiers ∀ LO ≤ n < HI|K(n), tuple formers [LO ≤ n < HI|K(n)], and forall iterators (∀ LO ≤ n < HI|K(n)) *block*, all these operations depending on range iterators.

D. Differentiation of Expressions Containing Parameters on which They Depend Discontinuously.

Most SETL expressions are not continuous in all the parameters on which they depend. For example, the set former

(1) $C = \{x \in s \mid f(x,q) > q\}$

is continuous in the set s and the mapping f, but it is discontinuous relative to changes in the free variable q.

Suppose that the expression (1) occurs in a strongly connected region R, and suppose also that all changes to s and f are slight within R. In this situation the difficulties caused by the discontinuous dependence of (1) on q can be overcome, and the computation (1) can be moved out of R by applying the general formal differentiation scheme sketched in the introduction to the present thesis. That is, we can perform the following steps:

i. Define an initial mapping $\bar{C}$:= *nullset* on entrance to R.

ii. Replace all computations (1) in R by the expression

if q ∈ *DOM* $\bar{C}$ *then* $\bar{C}(q)$ *else* $\bar{C}(q)$:= {x ∈ s|f(x,q) > q}

which either retrieves the value of a stored calculation of (1) from $\bar{C}$ if such a value exists or else computes (1) and records this value into the memo function $\bar{C}(q)$ for possible future use.

iii. Whenever differential changes to s or f occur in R, modify each stored set $\bar{C}(q)$ according to rules 1 and 2 (cf. section C) for all values q ∈ *DOM* $\bar{C}$; i.e., execute the following prederivative code just before s := s $\pm$ Δ:

(2) (∀q ∈ *DOM* $\bar{C}$) $\bar{C}(q)$:= $\bar{C}(q)$ $\pm$ {x ∈ Δ|f(x,q) > q};

The basic pre and postderivatives which keep $\bar{C}$ available on exit from indexed assignments $f(x_1,x_2)$:= Z are as follows:

(3) $(\forall q \in DOM\ \bar{C})\ s_0(q) := \{x \in s \mid x = x_1\ \&\ q = x_2\};$

$\bar{C}(q) := \bar{C}(q) - \{x \in s_0(q) \mid f(x,q) > q\};$

end ∀;

$f(x_1,x_2) := Z$

$(\forall q \in DOM\ \bar{C})\ \bar{C}(q) := \bar{C}(q) + \{x \in s_0(q) \mid f(x,q) > q\}$

end ∀;

The case (1) deserves special treatment, since the sets $s_0(q)$ appearing in (3) can be calculated by (9) of Section(C); i.e., $s_0(q)$:= if $x_1 \in s\ \&\ q = x_2$ *then* $\{x_1\}$ *else nullset* where x_1 and x_2 need not be region constants. As a result, (3) can be transformed into a speedier equivalent version,

(3') *if* $x_1 \in s\ \&\ x_2 \in DOM\ \bar{C}\ \&\ f(x_1,x_2) > x_2$ *then*

$\bar{C}(x_2) := \bar{C}(x_2) - \{x_1\};$

endif;

$f(x_1,x_2) := Z;$

if $x_1 \in s\ \&\ x_2 \in DOM\ \bar{C}\ \&\ Z > x_2$ *then*

$\bar{C}(x_2) := \bar{C}(x_2) + \{x_1\};$

endif;

iv. For changes of q in R nothing more is needed.

The approach described by (i)-(iv) above will be profitable when the execution frequency of the code inserted by (ii) is great enough and when the maximum number $\#\bar{C}$ of calculations that need to be stored in $\bar{C}$ is small enough. But if $\#\bar{C}$ is large three major objections which can easily make this approach infeasible arise:

(a) storage of all the sets $\bar{C}(q)$ may be too expensive;

(b) updating all the sets $\bar{C}(q)$ when a parameter upon which C depends continuously is modified may waste more time than is saved by avoiding the calculation of C;

(c) storage of the domain *DOM* $\bar{C}$ of $\bar{C}$ may be too expensive.

Example (1) illustrates these three potential objections. For a 'randomly' chosen f some large percentage of all the $x \in s$ will belong to the set (1) for many q. Hence, the sets $\bar{C}(q)$ will be large for many q. These sets will often overlap and when $\#\bar{C}$ is large storing them will undoubtedly require much more space than is required by s. It is not difficult to surmise that in those contexts where objection (a) arises objection (b) will also cause trouble. Although the update code (3') and the computations introduced in step ii above are inexpensive, the work involved in computing the derivative (2) is directly proportional to $\#\bar{C}$ and can exceed the cost of the original calculation (1) for large $\#\bar{C}$. The third objection (c) will arise when excessive space is needed for storing the domain of $\bar{C}$ and when q is a set or tuple valued variable occurring in (1).

If we consider a general expression $C = E(x_1,\ldots,x_k,x_{k+1},\ldots,x_n)$ that depends discontinuously in R on changes to its free variables $x_{k+1},\ldots,x_n$ then objections (a), (b) and (c) can be extremely difficult to overcome. The basic formal differentiation method sketched

in the Introduction requires that we use a map $\bar{C}$ to store separate values of expressions E' instantiating E in appropriately chosen values of $x_{k+1},\ldots,x_n$. If for $i = k+1,\ldots,n$ we let D_{x_i} refer to the range of values that x_i can have in R, then the maximum number of stored calculations in $\bar{C}$ is $\prod_{i=k+1}^{n} \#D_{x_i}$. Even if each discontinuity parameter x_i is boolean valued, $\bar{C}$ can come to store as many as 2^{n-k} different calculations.

One possible way of diminishing the number of values that need to be stored in such cases is to group the parameters $x_{k+1},\ldots,x_n$ into subexpressions e of E which depend only on $x_{k+1},\ldots,x_n$. The advantage of this rests on an elementary property of finite mappings: the cardinality of the range of such a mapping never exceeds the size of its domain. We can use this approach when each subexpression e of E behaves like a finite map; i.e., when E is *applicative*.

As an example of this, note that in dealing with an expression

$$(4)\quad C = \{x \in s \mid f(x + q_3) = (f(q_1 + g(q_2)) + q_3)\}\ ,$$

we can reduce the number of values of C that need to be stored by using a map $\bar{C}_1(q_3, f(q_1 + g(q_2)) + q_3)$ instead of $\bar{C}_2(f,g,q_1,q_2,q_3)$.

To reduce (4) to the map $\bar{C}_1$, we can take the following steps:

(i) On entrance to R, perform the initialization

$\bar{C}$:= *nullset*.

(ii) Within R replace all calculations of C by uses of the expression

if $[b_1,b_2] := [q_3, f(q_1+g(q_2))+q_3] \in$ PROJECT(2,$\bar{C}$)

then $\bar{C}(b_1,b_2)$

else $\bar{C}(b_1,b_2) := \{x \in s | f(x + b_1) = b_2\}$.

(iii) At each program point in R at which s undergoes a 'slight' change s := s $\pm$ Δ , execute the following prederivative code,

(5) (∀$[b_1,b_2] \in$ PROJECT(2,$\bar{C}$)) $\bar{C}(b_1,b_2) := \bar{C}(b_1,b_2)$

$\pm \{x \in \Delta | f(x + b_1) = b_2\}$;

END ∀;

which is justified by Rule 1. Note that (5) can be rewritten in an efficient form as follows:

(5') (∀$x \in (\Delta \bar{*} s)$, $b_1 \in$ DOM $\bar{C} | b_2 := f(x+b_1) \in$ *DOM* $\bar{C}\{b_1\}$)

$\bar{C}(b_1,b_2) := \bar{C}(b_1,b_2) \pm \{x\}$;

END ∀;

The cost of (5) is directly proportional to $\#\bar{C} \times \#\Delta$ while the cost of (5') is $O(\#DOM\ \bar{C} \times \#\Delta)$ where *DOM* $\bar{C}$ is the set of first parameter values of $\bar{C}$.

(iv) Whenever an indexed assignment f(y) := Z occurs in R, we can keep $\bar{C}$ available by use of Rule 2; which is to say by executing the following code:

```
(6)  (∀[b1,b2] ∈ PROJECT(2,C̄)) s0(b1,b2) := {x∈s|x+b1=y};
          C̄(b1,b2) := C̄(b1,b2)
            - {x ∈ s0(b1,b2)|f(x+b1) = b2};
     END ∀;
     f(y) := Z;
     (∀[b1,b2]∈PROJECT(2,C̄))C̄(b1,b2) := C̄(b1,b2)+{x∈s0(b1,b2)|
                                          f(x+b1) = b2};
     END ∀;
```

This can be further optimized to yield the following code,

```
(6') (∀b1 ∈ DOM C̄)
          if y-b1 ∈ s & b2 := f(y) ∈ DOM C̄{b1}
              then C̄(b1,b2) := C̄(b1,b2) - {y-b1};
          endif
     end ∀;

     f(y) := Z;
     (∀b1 ∈ DOM C̄)
          if y-b1 ∈ s & Z ∈ DOM C̄{b1}
              then C̄(b1,Z) := C̄(b1,Z) + {y-b1};
          endif
     end ∀;
```

It is easy to see that the computational cost of (6') is proportional to #*DOM* $\bar{C}$. Note that y need not be a region constant in (6').

(v) In keeping $\bar{C}$ available in R, we can ignore all modifi-

cations to the map g or to the free variables q_1, q_2, or q_3 of (4). We note that by making use of a map $\bar{C}_1$ instead of the more straightforward $\bar{C}_2$ to reduce (4), the method of parameter regrouping just described excludes values of the sets f and g which are part of the domain of $\bar{C}_2$ from the domain of $\bar{C}_1$ and thereby ameliorates objection (c) as well as objections (a) and (b).

Although the last example illustrates some advantages of 'regrouping', the transformations of (5) to (5') and (6) to (6') go beyond 'regrouping' and illustrate more powerful improvement techniques. In the remainder of this section we will utilize these techniques systematically to extend Rules 1 and 2 to additional methods for reducing generalized set formers

$$(7) \qquad C = \{x \in s \mid K(x,q_1,q_2,\ldots,q_n)\} .$$

This expression depends continuously on differential changes to s and on indexed assignments to mappings f occurring in K (provided that at least one such occurrence of each such f contains a parameter expression involving the bound variable x). However, (7) depends discontinuously on changes in the free variables $q_1,\ldots,q_n$.

Suppose that we want to reduce (7) in a strongly connected region R, and suppose that values of (7) are stored in a map $\bar{C}(q_1,\ldots,q_n)$. Then within R, and for all changes of s by the operation $s := s \pm \Delta$, the following prederivative code can be generated by a straightforward

application of Rule 1:

(8) $(\forall[q_1,\ldots,q_n] \in \text{PROJECT}(n,\bar{C}))$

$\bar{C}(q_1,\ldots,q_n) := \bar{C}(q_1,\ldots,q_n) \pm \{x\in\Delta \mid K(x,q_1,\ldots q_n)\};$

END ∀;

Inspection of (8) leads us to anticipate a time cost directly proportional to $\#\bar{C}\times\#\Delta\times\text{Cost}(K(x,q_1,\ldots,q_n))$. To update (7) at indexed assignments $f(y_1,\ldots,y_m) := Z$ for which the f terms involved in x and occurring in K are $f(p_{i1}(x,q_1,\ldots,q_n),\ldots,p_{im}(x,q_1,\ldots,q_n))$ for $i = 1,\ldots,r$ the pre and post derivative code generated by extending the rule (5) of section C (i.e., the rule from which Rule 2 is derived) would be as follows:

(9) $(\forall[q_1,\ldots,q_n] \in \text{PROJECT}(n,\bar{C}))$

$s_0(q_1,\ldots,q_n) := \{x\in s \mid \underset{i=1}{\overset{r}{\text{OR}}}\left(\underset{j=1}{\overset{m}{\&}} p_{ij}(x,q_1,\ldots,q_n)=y_j\right)\}$

$\bar{C}(q_1,\ldots,q_n) := \bar{C}(q_1,\ldots,q_n) - \{x\in s_0(q_1,\ldots q_n) \mid K(x,q_1,\ldots,q_n)\};$

END ∀;

$f(y_1,\ldots,y_m) := Z;$

$(\forall[q_1,\ldots,q_n] \in \text{PROJECT}(n,\bar{C}))$

$\bar{C}(q_1,\ldots,q_n) := \bar{C}(q_1,\ldots,q_n) + \{x\in s_0(q_1,\ldots,q_n) \mid K(x,q_1,\ldots,q_n)\};$

END ∀;

This code can sometimes be modified in the manner described in the last section (cf. 2.C Ex. (9)-(11)) to yield code which keeps s_0 available in R via incremental 'easy' calculations.

It is not difficult to see that computation of (8) or (9) can be unacceptably expensive due to objections (a) and (b). We also note that simple regrouping of discontinuity parameters need not always remove these objections successfully.

Nevertheless, these problems can be overcome in cases in which $\bar{s}_0$ and $\bar{C}$ are continuous relative to differential changes in the continuity parameters of (7). Fortunately, this holds for a few special cases of common occurrence in SETL programs. These special cases which may be said to involve 'removable' discontinuities include set formers based on the elementary forms,
{x ∈ s | f(x)}, {x ∈ s | f(x) = q}, {x ∈ s | f(x) ∈ Q},
{x ∈ s | f(x) ∉ Q}, {x ∈ s | q ∈ f(x)}, and
{x ∈ s | f(x) <relop> q} (where <relop> can be $<, \leq, >, \geq$). Set formers involving boolean valued subexpressions which include terms q ∉ f(x) and f(x) ¬= q are not directly susceptible to reduction.

In the following discussion we will show how to reduce the basic set formers above and more general set expressions built up from these by combining boolean valued subexpressions of these elementary forms using the logical connectives &, *or*, and ¬ . We will also present techniques for handling still more general set formers containing unfavorable relational operations within their boolean subexpressions.

A most important special case is

(10) $\qquad C = \{x \in s \mid f(x) = t\}$.

To differentiate (10), where we assume that the values of (10) are stored in a map $\bar{C}(t)$, we can apply Rule 1 and are led to the prederivative

(11) $(\forall q \in DOM\ \bar{C})\ \bar{C}(q) := \bar{C}(q) \pm \{x \in \Delta \mid f(x) = q\};$
END $\forall$;

of (10) with respect to $s := s \pm \Delta$.

We know from the preceding discussion of objection (b) that if $\#DOM\ \bar{C}$ is too large the computation (11) will be costlier than a full recalculation of (10). However, it is actually only necessary to apply Rule 1 to those sets $\bar{C}(q)$ that actually change. But $\bar{C}(q)$ will not change if $\{x \in \Delta \mid f(x) = q\}$ is empty. Thus, the set $DOM\ \bar{C}$ appearing in (11) can be replaced by

(12) $\qquad \{q \in \mathrm{DOM}\ \bar{C} \mid (\exists x \in \Delta \mid f(x) = q)\}$

which is usually a smaller set than $DOM\ \bar{C}$. (But (12) may be costlier to compute.) Moreover, (12) can be transformed into an 'easier' calculation:

(12') $\qquad \{f(x) : x \in \Delta \mid f(x) \in DOM\ \bar{C}\}$

which computes the same set as (12) but in $O(\#\Delta)$ time. Also, since redundant calculations $\bar{C}(q) := \bar{C}(q) \pm \{x \in \Delta \mid f(x) = q\}$ of (11) can be introduced without invalidating (11), the set (12') need never be computed at all; we can simply rewrite (11) as

(13) $(\forall x \in \Delta | f(x) \in DOM\ \bar{C})\ \bar{C}(f(x)) :=$

$$\bar{C}(f(x)) \pm \{y \in \Delta | f(y) = f(x)\};$$

END ∀;

Finally after transforming the repeated assignment in (13) to the equivalent

(14) $(\forall y \in \Delta | f(y) = f(x))\ \bar{C}(f(x)) := \bar{C}(f(x)) \underline{+} \{y\};$

END ∀;

and jamming the iterator of (14) with (13) we arrive at a highly efficient prederivative

(15) $(\forall x \in \Delta | f(x) \in DOM\ \bar{C})\ \bar{C}(f(x)) := \bar{C}(f(x)) \underline{+} \{x\};$

END ∀;

equivalent to (13) but requiring only $O(\#\Delta)$ elementary steps. Note here that we have demonstrated the correctness of (15) informally by first noting a conceptually easy but inefficient metatransformation (8) and then by passing to (11) as a particular but still inefficient instance of (8). The form (15) is then derived easily using simplifying transformations. We will use this approach repeatedly throughout this chapter.

If the map f of (10) is changed within L by an indexed assignment f(y) := Z then since the f depth of (10) is 1 and because the boolean subpart of $s_0 = \{x \in s | x = y\}$ is linear in x, and does not depend on t, the general derivative computation (9) leads to the following version of Rule 2:

```
(16)   (∀q∈ DOM C̄)
          s0 := {y} * s;
          C̄(q) := C̄(q) - {x ∈ s0|f(x) = q};
       END ∀;
       f(y) := Z;
       (∀q ∈ DOM C̄) C̄(q) := C̄(q) + {x ∈ s0|f(x) = q};
       END ∀;
```

Since in (16) s_0 is a compiler generated variable used only once, since the expression {y} * s is invariant in (16), and since s_0 will be useless on exit from (16), we can remove the single assignment to s_0 and replace all uses of s_0 in (16) by occurrences of {y} * s. We can also apply the chain of transformations (11)-(15) to the iterators in (16). This leads to the following improved code:

```
(16')  (∀x ∈ {y} * s | f(x) ∈ DOM C̄)
          C̄(f(x)) := C̄(f(x)) - {x};
       END ∀;
       f(y) := Z;
       (∀x ∈ {y} * s | f(x) ∈ DOM C̄)
          C̄(f(x)) := C̄(f(x)) + {x};
       END ∀;
```

We can also rearrange the iterators in (16') into the form (∀x ∈ {y}|x ∈ s & f(x) ∈ *DOM* C̄)<BLOCK(x)> which simplifies immediately to *if* y ∈ s & f(y) ∈ *DOM* C̄ *then* <BLOCK(y)>.

Our final Rule 2 variant is then

```
(16")  if y ∈ s & f(y) ∈ DOM C̄ then C̄(f(y)) := C̄(f(y))-{y};
       endif;
       f(y) := Z;
       if y ∈ s & f(y) ∈ DOM C̄ then C̄(f(y)) := C̄(f(y))+{y};
       endif;
```

which can be computed in essentially constant time.

According to our standard method, we can differentiate an expression (10) in a region R by inserting the code (15) and (16") at appropriate program points within R, inserting $\bar{C}$:= *nullset* on entrance to R, and replacing uses of (10) in R by

```
(17)  if t ∈ DOM C̄ then C̄(t) else C̄(t) := {x∈s|f(x) = t}.
```

But special characteristics of (10) can sometimes be exploited to obtain several further improvements. It is instructive to note that if we could determine at compile time the range D_t of values that the discontinuity parameter t can have in R, then on entrance to R we could store values of (10) in $\bar{C}$ for all values of t in D_t. This could be done by using the following efficient code,

```
(18)      (∀q ∈ D_t) C̄(q) := nullset; end ∀;
          (∀x ∈ s | f(x) ∈ D_t)
                C̄(f(x)) := C̄(f(x)) + {x};
          end ∀;
```

which computes the same thing as

```
C̄ := nullset;
(∀q ∈ D_t) C̄(q) := {x ∈ s | f(x) = q};
end ∀;
```

Consequently uses of (10) in R could be replaced by the simple retrieval operation $\bar{C}(t)$.

Since the method just described does not involve a 'memo' function, its overall computational cost is easy to predict. Only the inexpensive calculations (15) and (16") are introduced into R, while the cost of (18) is about the same as (10) which we eliminate from R. Thus, we expect a speedup considerably greater than that attained by the previous method which was complicated by overhead in memo function maintenance operations such as (17).

Unfortunately, the method we have just described is not generally applicable since it requires determination of the set D_t, undecidable at compile time. Nevertheless, there is another similar approach which offers similar speedup advantages over the standard method, but does not depend on any difficult analysis. This new approach essentially replaces D_t by the image set f[s] (image of f restricted to s); this set includes all values of q for which $\bar{C}(q) = \{x \in s | f(x) = q\}$ is nonnull. Whenever f[s] is spoiled by changes in f and s, $\bar{C}$ can record new values in memo function fashion, and it can also undergo differential modifications within derivative code. Consequently we are able to remove all calculations of (10) from R and

replace them by *if* t ∈ *DOM* $\bar{C}$ *then* $\bar{C}$(t) *else* *nullset*.

The following additional steps describe this method more fully.

i. On entrance to R, initialize $\bar{C}$ by executing

```
(19)    C̄ := nullset;
        (∀x ∈ s) if f(x) ∈ DOM C̄ then
                              C̄(f(x)) := C̄(f(x)) + {x}; else;
                              C̄(f(x)) := {x}; endif;
```

which costs about the same as (10) to execute.

ii. Instead of (15), use the following variants.

```
(20)    /* for s := s + Δ */
        (∀x ∈ (Δ - s)) if f(x) ∈ DOM C̄ then
             C̄(f(x)) := C̄(f(x)) + {x}; else
             C̄(f(x)) := {x};;;
```

and

```
(20')   /* for s := s - Δ */
        (∀x ∈ (Δ * s)|f(x) ∈ DOM C̄)
             C̄(f(x)) := C̄(f(x)) - {x};;
```

respectively as prederivatives of (10) relative to s := s$\pm$Δ. Note that *DOM* $\bar{C}$ can change in (20) and (20').

iii. After inserting into (16") code which updates *DOM* $\bar{C}$, we can use the following derivative for indexed assignments to f,

```
(21)    if y∈s & f(y)∈DOM C̄ then C̄(f(y)) := C̄(f(y))-{y};
        endif;
        f(y) := Z
        if y ∈ s then
             if f(y) ∈ DOM C̄ then C̄(f(y)) := C̄(f(y))+{y} else
                                  C̄(f(y)) := {y};
             endif;
        endif;
```

It should be clear that the transformations just described can attain greater speedup than the standard technique. What's more, when $\#D_t$ is large, a better utilization of space is also achieved.

The preceding results apply in an interesting way to a class of set formers typified by

$$C = \{x \in s \mid f(x) \in q\} \quad , \tag{22}$$

where the free variable q is a set. Recall from section (C) that (22) is continuous relative to small changes in s and to indexed assignments to f. If q is changed by a computation $q = q \pm \Delta$ where $\#\Delta << \#q$, then the corresponding prederivative

$$C = C \pm \{x \in s \mid f(x) \in \Delta\} \tag{23}$$

will often represent a small change to C. However, because (23) still requires an iteration over s, this update computation will often be too expensive to allow profitable

reduction of (22).

For this reason, it is appropriate in handling (22) to use the identity

$$\{x \in s \mid f(x) \in \Delta\} = \bigcup_{b \in \Delta} \{x \in s \mid f(x) = b\} .$$

The expressions $C' = \{x \in s \mid f(x) = b\}$ which then appear can be differentiated by the methods sketched earlier in the present section; i.e., by using the following code as a first prederivative of (22) with respect to $q := q \pm \Delta$,

```
(24)        (∀y ∈ Δ, w ∈ {u ∈ s | f(u) = y})
                C := C ± {w};
            end ∀;
```

But to handle (24) efficiently we must formally differentiate $C' = \{u \in s \mid f(u) = y\}$ and store its values as a map $\bar{C}'(y)$.

Set formers like

$$C = \{x \in s \mid f(x) \notin q\} \tag{25}$$

can be treated similarly. A prederivative of (25) with respect to $q := q \pm \Delta$ is

$$C := C \mp \{x \in s \mid f(x) \in \Delta\} \tag{26}$$

which leads at once to a more efficient prederivative

```
(26')       (∀y ∈ Δ, w ∈ {u ∈ s | f(u) = y})
                C := C ∓ {w};
            END ∀;
```

Formula (26') can then be improved in much the same way as (24).

Set formers involving boolean valued subexpressions based on comparison operations such as

(27) $$C_1 = \{x \in s \mid f(x) < q\}$$

can be treated as special cases of (22). To see this, let M be the largest q value that needs to be considered, and let m be the minimum value of $\{f(x), x \in s\}$ over all f and s that can appear. Putting $sq := \{b, m \leq b < q\}$, we see that (27) is equivalent to $\{x \in s \mid f(x) \in sq\}$.

If for $\Delta > 0$, q changes slightly by $q := q \pm \Delta$, then sq changes, also slightly, by

```
sq := sq +{b: q ≤ b < q + Δ} /* for q := q + Δ */
```

or by

```
sq := sq - {b: q - Δ ≤ b < q}  /* for q := q-Δ */
```

Thus, to update C_1 we can simply execute the prederivative code

```
(28)  (∀q ≤ y < q+Δ, w ∈ {u ∈ s|f(u) = y})  /* for q:=q+Δ */
          C1 := C1 + {w};
      end ∀;
      or
      (∀q > y ≥ q-Δ, w ∈ {u ∈ s|f(u) = y})  /* for q:=q-Δ */
          C1 := C1 - {w};
      end ∀;
```

which can be further improved by differentiating the set formers $\bar{C}'(y) = \{u \in s \mid f(u) = y\}$ as in the last two examples.

However, the total ordering T = (<, *DOM* $\bar{C}'$) can be exploited to further optimize the iteration that appears within (28) by techniques not generally applicable to (24) To do this, on entrance to R we sort *DOM* $\bar{C}'$ in ascending order and produce the predecessor and successor maps, *pred* and *succ*, on T (where pred(Ω) is the maximum element of *DOM* $\bar{C}'$). Next, we make the assignment k := [*min*: w ∈ *DOM* $\bar{C}' \mid \neg(w < q)]$w. Then we can keep C_1, *pred*, *succ*, and k available collectively in R by efficient incremental calculations. Thus, whenever q is changed 'slightly' by q := q ± Δ, the following efficient prederivatives can be executed,

```
(28')   (while k < q+Δ)              /* for q := q+Δ */
            C1 := C1 + C̄'(k);
            k  := succ(k);
        end while;
```

and

```
(28")   (while pred(k)> q-Δ)         /* for q := q-Δ */
            k := pred(k);
            C1:= C1 - C̄'(k);
        end while;
```

Note that (28') and (28") can run considerably faster than (28) when *DOM* $\bar{C}'$ is sparse in the interval [m,M]. Whenever

s undergoes a change $s := s \pm \Delta$, and if *DOM* $\bar{C}'$ is maintained as a balanced tree, then the appropriate insertions and deletions necessary to maintain *pred*, *succ*, and k, require $\#\Delta \times \log (\#DOM\ \bar{C}')$ steps.

Another class of special cases derives from

(29) $$C = \{x \in s \mid q \in f(x)\}$$

a set former which despite its close resemblance to (22) must be handled very differently. While (22) is continuous in all of its parameters, (29) is discontinuous in q. Thus we must save the value of C in a map $\bar{C}(q)$. Fortunately, however, the discontinuity which appears here is removable. If we apply the general rule (8) to (29) we obtain prederivative code

(30) $(\forall t \in DOM\ \bar{C})\ \bar{C}(t) := \bar{C}(t) \pm \{x \in \Delta \mid t \in f(x)\};$ end ∀;

for modifications $s := s \pm \Delta$. This code can be improved by extending the iteration not over all of *DOM* $\bar{C}$ but over the smaller set

$$C' = \{t \in DOM\ \bar{C} \mid (\exists x \in \Delta \mid t \in f(x))\}$$

which can be written equivalently as

$$C' = [+: x \in \Delta]\ f(x) * DOM\ \bar{C}\ .$$

Further symbolic manipulation of (30) leads to the following prederivative ,

(30') $(\forall x \in \Delta,\ t \in f(x) * DOM\ \bar{C})\ \bar{C}(t) := \bar{C}(t) \pm \{x\};\ end\ \forall;$

which is generally preferable to (30) (especially when $\#f(x) << \#DOM\ \bar{C}$).

If f undergoes a change f(y) := Z, then since $s_0 = \{x \in s | x = y\}$ does not depend on q and can be calculated efficiently on-the-fly, the general derivative formula (9) realizes the following Rule 2 update computation,

(31)
s_0 := *if* y ∈ s *then* {y} *else nullset*;
(∀t ∈ *DOM* $\bar{C}$)
$\bar{C}(t) := \bar{C}(t) - \{x \in s_0 | t \in f(x)\}$;
END ∀;
f(y) := Z;
(∀t ∈ *DOM* $\bar{C}$) $\bar{C}(t) := \bar{C}(t) + \{x \in s_0 | t \in f(x)\}$
END ∀;

However, it is not difficult to see that (31) can be rewritten more efficiently as

(31')
if y ∈ s *then* (∀u ∈ f(y) * *DOM* $\bar{C}$)
$\bar{C}(u) := \bar{C}(u) - \{y\}$; *end* ∀;
endif;
f(y) := Z
if y ∈ s *then* (∀u ∈ Z * *DOM* $\bar{C}$)
$\bar{C}(u) := \bar{C}(u) + \{y\}$; *end* ∀;
endif;

Finally, note that a suitable prederivative of (29) with respect to the change f(y) := f(y) $\pm$ Δ of f is given by the following special case of (31'):

(32) *if* y $\in$ s *then* ($\forall$u $\in$ Δ * *DOM* $\bar{C}$)

$\bar{C}$(u) := $\bar{C}$(u) $\pm$ {y}; *end* $\forall$;

endif;

Since the derivative rules (30'), (31') and (32) conform to our standard reduction method, in order to differentiate (29) fully we only need to initialize $\bar{C}$ on entrance to R by executing $\bar{C}$:= *nullset* and to replace all uses of (29) in R by the calculation,

(33) *if* q $\in$ *DOM* $\bar{C}$ *then* $\bar{C}$(q) *else* $\bar{C}$(q) := {x$\in$s | q$\in$f(x)}.

However, by introducing (33), we fail to eliminate all calculations of (29) from R; this imperils our chance of attaining a worthwhile speedup. Nevertheless, we can improve the handling of the memo function $\bar{C}$ for (29) as we did in the previous special case (10), and can effectively eliminate all uses of (29) from R.

To do this, note first of all that values of (29) stored in $\bar{C}$(q) are only meaningful (i.e. nonnull) when q is in the range $Q = \bigcup_{x \in s} f(x)$ of f. Thus, by keeping the restriction of $\bar{C}$ to Q available in R we can replace all uses of (29) by the conditional expression, *if* q $\in$ *DOM* $\bar{C}$ *then* $\bar{C}$(q) *else nullset*. The following steps achieve this result:

i. On entrance to R execute

```
C̄ := nullset;
(∀x ∈ s, t ∈ f(x))
     if t ∈ DOM C̄ then C̄(t) := C̄(t) + {x};
                  else  C̄(t) := {x};
     endif;
end ∀;
```

which takes O(#f) steps to compute but does the same thing as

```
C̄ := nullset;
(∀x ∈ s, t ∈ f(x)) C̄(t) := {y ∈ s|t ∈ f(y)};
end ∀;
```

ii. At points in R where s changes, execute the appropriate prederivative, which will be either

```
(∀x ∈ (Δ - s), t ∈ f(x))        /* for s := s+Δ */
     if t ∈ DOM C̄ then C̄(t) := C̄(t) + {x};
                  else C̄(t) := {x};
     endif;
end ∀;
```

or

```
/* for s := s -Δ */
(∀x ∈ (Δ * s), t ∈ (f(x) * DOM C̄)) C̄(t) := C̄(t) - {x};
end ∀;
```

(This is based on (30').) When, #f(x) is uniformly bounded by δ, and when $\delta << \#s$, then the cost of these last

calculations is $O(\delta \times \#\Delta)$ which we expect to be inexpensive relative to (29).

iii. For changes f(y) := Z, we can use the following derivative code.

```
if y ∈ s then (∀u ∈ f(y) * DOM C̄)
                    C̄(u) := C̄(u) - {y};
                end ∀;
endif;
f(y) := Z;
if y ∈ Z then (∀u ∈ Z) if u ∈ DOM C̄ then
                    C̄(u) := C̄(u) + {y}; else
                    C̄(u) := {y}; endif;
                end ∀;
endif;
```

which executes in no more than $O(\delta)$ elementary steps.

iv. The rule corresponding to (32) for updating $\bar{C}$ with respect to f(y) := f(y) $\pm$ Δ is

```
/* for f(y) := f(y) + Δ */
if y ∈ s then (∀u ∈ (Δ - f(y))) if u ∈ DOM C̄ then
                    C̄(u) := C̄(u) + {y}; else
                    C̄(u) := {y}; endif;
                end ∀;
endif;
```

and

```
/* for f(y) := f(y) - Δ */
if y ∈ s then (∀u ∈ (Δ* f(y) * DOM C̄))
                    C̄(u) := C̄(u) - {y};
                end ∀;
endif;
```

both of these requiring only O(#Δ) steps.

It is not difficult to predict with some assurance that the method just proposed offers a better chance of speedup than does the standard method.

The two expressions $C_1 = \{x \in s | f(x) \neg= q\}$ and $C_2 := \{x \in s | q \notin f(x)\}$ can be handled by transforming them into the more convenient forms,

$$s - \{x \in s \mid f(x) = q\}$$

and

$$s - \{x \in s \mid q \in f(x)\}$$

respectively. The setformers $\{x \in s | f(x) = q\}$ and $\{x \in s | q \in f(x)\}$ thus exposed can usually be reduced profitably by methods previously described.

Each of these examples (10), (22), (25), (27), and (29) typifies the treatment of a broad class of expressions that can often be differentiated profitably. Within the class associated with (10) we consider the set formers

$$C = \{x \in s \mid K_1(x) = K_2(q_1,\ldots,q_n)\} , \tag{34}$$

where $q_1,\ldots,q_n$ are free variables upon which C depends discontinuously. We assume that K_1 of (34) is a subexpression

only involving x, parameters upon which (34) depends continuously, and maps f_i upon which C can depend discontinuously but whose occurrences in K_1 all have parameters depending on x. K_2 of (34) is assumed to be a subexpression only involving the parameters $q_1,\ldots,q_t$ on which C depends discontinuously, and the maps f_i.

We can treat (34) easily if we recognize that it is constructed from (10) by a composition in which K_1 replaces f and $K_2(q_1,\ldots,q_n)$ replaces t. To formally differentiate (34) in a program region R we can store separate calculations (34) in a map $\bar{C}(K_2(q_1,\ldots,q_n))$ proceeding in the following steps:

i. On entrance to R perform

```
C̄ := nullset;
(∀x ∈ s) if K1(x) ∈ DOM C̄ then
     C̄(K1(x)) := C̄(K1(x))+ {x}; else
     C̄(K1(x)) := {x};
     endif;
end ∀;
```

(This is based on (19)).

ii. Whenever s changes by $s := s \pm \Delta$, execute the corresponding prederivative code,

```
(35)   (∀x ∈ (Δ - s)) if K1(x) ∈ DOM C̄ then
                          C̄(K1(x)) := C̄(K1(x))+ {x}; else
                          C̄(K1(x)) := {x};
                       endif;
       end ∀;
```

for s := s + Δ and

(35') (∀x ∈ (Δ * s)|$K_1(x)$ ∈ *DOM* $\bar{C}$)

$\bar{C}(K_1(x))$:= $\bar{C}(K_1(x))$ - {x};

end ∀;

for s := s - Δ. (Both (35) and (35') can be derived from (20) and (20') by substituting K_1 for f.)

iii. Modifications (in R) of $q_1,\ldots,q_n$ or any map f_i occurring in K_2 (but not in K_1 of (34)) do not require insertion of update code.

iv. Replace all occurrences of (34) in R by occurrences of

if $K_2(q_1,\ldots,q_n)$ ∈ *DOM* $\bar{C}$ *then* $\bar{C}(K_2(q_1,\ldots,q_n))$ *else nullset*.

In constructing a derivative of (34) with respect to indexed assignments $f(y_1,\ldots,y_n)$:= Z, we cannot simply make use of the update code (21) used for example (10), because in deriving (21) we make use of characteristics of (10) absent in (34). In fact, we cannot easily overcome the inadequacies of our current formulation of Rule 2 to make it apply usefully here. Hence, before giving the desired derivative code for (34), we must iron out the lingering difficulties of Rule 2. This we will do shortly, after first studying the continuity properties of an important generalization of (34).

It is possible to differentiate more general set formers than the elementary ones just described by combining the boolean subparts of these forms using logical connectives

&, *or*, ¬ . Consider, as an example, the following setformer,

$$(36)\quad \{x \in s \mid \&_{i=1}^{m} (K_i(x) = K_i'(q_1,\ldots,q_n)) \ \&\ K_{m+1}(x)\}$$

in which for i = 1,...,m+1 the terms K_i are restricted in the same way as K_1 of (34) and for i = 1,...,m the terms K_i' are defined similarly to K_2 of (34). If we use a map $\bar{C}(K_1'(q_1,\ldots,q_n),\ldots,K_m'(q_1,\ldots,q_n))$ to store the values of (36), and if we treat $\bar{C}$ in a manner similar to the improved techniques used for handling (34), then the prederivatives of $\bar{C}$ with respect to the changes $s := s \pm \Delta$ can be written as

```
(37)    (∀x ∈ (Δ - s)|K_{m+1}(x))     /* for s := s+Δ */
           if [K_1(x),...,K_m(x)] ∈ PROJECT(m,C̄) then
              C̄(K_1(x),...,K_m(x)) := C̄(K_1(x),...,K_m(x))+{x};
              else C̄(K_1(x),...,K_m(x)) := {x};
           endif;
        end ∀;
```

and

```
        /* for s := s - Δ */
(37')   (∀x ∈ (Δ * s)|K_{m+1}(x) & [K_1(x),...,K_m(x)]∈PROJECT(m,C̄))
           C̄(K_1(x),...,K_m(x)) := C̄(K_1(x),...,K_m(x)) - {x};
        end ∀;
```

which generalize (35) and (35'). Note that if we consider the set t to be initially null then the prederivative code (37) of $\{x \in t \mid \&_{i=1}^{m} (K_i(x) = K_i'(q_1,\ldots,q_n)) \ \&\ K_{m+1}(x)\}$ with respect to a change $t := t + s$ gives the appropriate code

which, on entry to R stores initial values of (36) in $\bar{C}$.

Observe that if disjunctions rather than conjunctions occur in (36) then continuity will ordinarily fail. However, one important exception (cf. (7) of Section C)

$$(38)\quad \bar{s}_0(y_1,\ldots,y_m) = \{x \in s \mid or(\ \&_{i=1}^{r}\ p_{ij}(x) = y_j)\}_{j=1}^{m}$$

deserves special attention. In this case we can show that if the setformer (38) is continuous in all of its parameters except $y_1,\ldots,y_m$, then $\bar{s}_0$ is continuous in all of its parameters.

To differentiate (38) we take the following steps:

i. On entrance to R execute

$\bar{s}_0$:= *nullset*
($\forall x \in s,\quad 1 \leq i \leq r$)

if $[p_{i1}(x),\ldots,p_{ir}(x)] \in$ PROJECT$(r,\bar{s}_0)$ *then*

$\bar{s}_0(p_{i1}(x),\ldots,p_{ir}(x))$:= $\bar{s}_0(p_{i1}(x),\ldots,p_{ir}(x))$ + {x}; *else*

$\bar{s}_0(p_{i1}(x),\ldots,p_{ir}(x))$:= {x};

endif;

end $\forall$;

(This is based on (37) and a kind of 'redundant discontinuity parameter elimination'.) The cost of (39) is essentially the same as the cost of computing the set former (38).

ii. Within R, whenever s changes by $s := s \pm \Delta$ execute the prederivatives,

```
(40) (∀x ∈ (Δ-s), 1 ≤ i ≤ r)          /* for s := s+Δ */
       if[p_il(x),...,p_ir(x)] ∈ PROJECT(r,s̄_0) then
          s̄_0(p_il(x),...,p_ir(x)) := s̄_0(p_il(x),...,p_ir(x))+{x};else
          s̄_0(p_il(x),...,p_ir(x)) := {x};
       endif;
    end ∀;
```

and

```
(40') (∀x ∈ (Δ * s), 1 ≤ i ≤ r|[p_il(x),...,p_ir(x)]∈PROJECT(r,s̄_0))
          s̄_0(p_il(x),...,p_ir(x)) := s̄_0(p_il(x),...,p_ir(x))-{x};
     end ∀;
```

(These are similar to (37) and (37').) Note that the computational cost of either (40) or (40') is O(#Δ×r) steps, inexpensive relative to a calculation of the setformer (38).

iii. The following variant of (11) of Section C is a derivative of (38) with respect to indexed assignments to f:

```
(41) (∀x∈s̄_0(y_1,...,y_m), 1≤i≤r |
          [b_1,...,b_m] := [p_il(x),...,p_ir(x)] ¬= [y_1,...,y_m]
       &  [b_1,...,b_m] ∈ PROJECT(m,s̄_0))
       s̄_0(b_1,...,b_m) := s̄_0(b_1,...,b_m) - {x};
    end ∀;
    f(y_1,...,y_m) := Z;
    (∀x∈s̄_0(y_1,...y_m),1≤i≤r|[b_1,...b_m]:=[p_il(x),...p_ir(x)]
                                   ¬= [y_1,...,y_m])
       if [b_1,...,b_m] ∈ PROJECT(m,s̄_0) then
          s̄_0(b_1,...,b_m) := s̄_0(b_1,...,b_m) + {x}; else
          s̄_0(b_1,...,b_m) := {x};
       endif;
    end ∀;
```

If there exists a bound δ on the sizes of sets $s_0(y_1,\ldots,y_m)$ which is small relative to #s, then the cost of (41) which is proportional to $\delta \times r$ will be inexpensive relative to (38).

iv. Suppose that an n-ary mapping g which undergoes indexed assignments in R (and only such assignments) occurs in t different terms in (38), $g(q_{11}(x),\ldots,q_{1n}(x)),g(q_{t1}(x),\ldots q_{tn}(x))$. Then the derivative code (which keeps $\bar{s}_0$ current in R) associated with indexed assignments to g, is as follows,

(42) $(\forall x \in \bar{h}_0(v_1,\ldots,v_n),\ 1 \leq i \leq r \mid [p_{i1}(x),\ldots,p_{im}(x)] \in \text{PROJECT}(m,\bar{s}_0))$

$\bar{s}_0(p_{i1}(x),\ldots,p_{im}(x)) := \bar{s}_0(p_{i1}(x),\ldots,p_{im}(x)) - \{x\};$

end $\forall$;

$g(v_1,\ldots,v_n) := Z;$

$(\forall x \in \bar{h}_0(v_1,\ldots,v_n),\ 1 \leq i \leq r)$

if $[p_{i1}(x),\ldots,p_{im}(x)] \in \text{PROJECT}(m,\bar{s}_0)$ *then*

$\bar{s}_0(p_{i1}(x),\ldots,p_{im}(x)) := \bar{s}_0(p_{i1}(x),\ldots p_{im}(x)) + \{x\};$

else $\bar{s}_0(p_{i1}(x),\ldots,p_{im}(x)) := \{x\};$

endif;

end $\forall$;

where $\bar{h}_0(w_1,\ldots,w_n) = \{x \in s \mid \underset{i=1}{\overset{t}{or}} \left(\underset{j=1}{\overset{n}{\&}} q_{ij}(x) = w_j \right)\}$ is an auxiliary expression which must be differentiated in R in the same manner as (38). Note that (42), like (41) consists of 'easy' calculations.

The method just presented leads in an obvious way to inexpensive derivatives for expressions like (1) of Section C, and also for (34) and (36) of the present section. In fact, it supports a reformulation of Rule 2 which is both

more general and more efficient than that previously given (cf. (10) and (11) of Section C). This rule is stated as follows.

Rule 2. Consider general set formers (7) satisfying the following two conditions:

Condition 1. The discontinuity parameters $q_1,\ldots,q_n$ of (7) must be removable in the derivative (8); i.e., the size of the iteration in (8) must be bounded independently of $\#\bar{C}$, $\bar{C}$ being the mapping which stores separate values of (7).

Condition 2. Suppose that $f_1,\ldots,f_z$ are all those maps which change in R and which also begin occurrences of map retrieval terms involving x within the boolean subpart of (7). Then we require that for $i = 1,\ldots,z$, each change to f_i within R must be an indexed assignment and each f_i term which involves x within (7) must not also involve $q_1,\ldots,q_n$.

For $k = 1,\ldots,z$, we will denote the r_k distinguishable f_k terms satisfying Condition 2 above by

$$f_k(p^k_{i1}(x),\ldots,p^k_{im_k}(x)),\ \ldots,\ f_k(p^k_{r_k1}(x),\ldots,p^k_{r_km_k}(x)) .$$

Then to keep (7) available in R in the presence of indexed assignments to $f_1,\ldots,f_z$ and small changes to s, we must keep the auxiliary maps

$$\bar{s}_k(y_1,\ldots,y_{m_k}) = \{x \in s \mid \underset{i=1}{\overset{r_k}{or}}\ (\underset{j=1}{\overset{m_k}{\&}}\ p^k_{ij}(x) = y_j)\}$$

$k = 1,\ldots,z$ available also. Since these auxiliary maps are of the same form as (38), they can each be initialized on entrance to R by executing the code (39). They can also

be updated efficiently whenever s changes by using rules (40) or (40'). Finally, at indexed assignments $f_\ell(v_1,\ldots,v_{m_\ell}) := W$ we can combine the techniques of (41) and (42) to obtain the following derivative which updates $\bar{s}_1,\ldots,\bar{s}_z$, $\bar{C}$ collectively

(42') $(\forall x \in \bar{s}_\ell(v_1,\ldots,v_{m_\ell}))$

$(\forall 1 \leq i \leq r_1 \mid [p^1_{i1}(x),\ldots,p^1_{im_1}] \in \text{PROJECT}(m_1,\bar{s}_1))$

$\bar{s}_1(p^1_{i1}(x),\ldots,p^1_{im_1}(x)) := \bar{s}_1(p^1_{i1}(x),\ldots p^1_{im_1}(x)) - \{x\};$

end $\forall$;

$\vdots$

$(\forall 1 \leq i \leq r_\ell \mid [y_1,\ldots,y_{m_\ell}] := [p^\ell_{i1}(x),\ldots,p^\ell_{im_\ell}(x)] \in \text{PROJECT}(m_\ell,\bar{s}_\ell)$

& $[y_1,\ldots,y_{m_\ell}] \neq [v_1,\ldots,v_{m_\ell}]$)

$\bar{s}_\ell(p^\ell_{i1}(x),\ldots,p^\ell_{im_\ell}(x)) := \bar{s}_\ell(p^\ell_{i1}(x),\ldots p^\ell_{im_\ell}(x)) - \{x\};$

end $\forall$;

$\vdots$

$(\forall 1 \leq i \leq r_z \mid [p^z_{i1}(x),\ldots,p^z_{im_z}] \in \text{PROJECT}(m_z,\bar{s}_z))$

$\bar{s}_z(p^z_{i1}(x),\ldots,p^z_{im_z}(x)) := \bar{s}_z(p^z_{i1}(x),\ldots p_{im_z}(x)) - \{x\};$

/* INSERT derivative code at this point for */

/* updating $\bar{C}$ with respect to s := s - {x} */

end $\forall$;

$f_\ell(v_1,\ldots,v_\ell) := W;$

```
(∀x ∈ s̄_ℓ(v_1,...,v_{m_ℓ}))
  (∀1 ≤ i ≤ r_1)
     if [p^1_{i1}(x),...,p^1_{im_1}(x)] ∈ PROJECT(m_1,s̄_1) then
          s̄_1(p^1_{i1}(x),...,p^1_{im}(x)) :=
                    s̄_j(p^1_{i1}(x),...,p^1_{im_1}(x)) + {x}; else
          s̄_1(p^1_{i1}(x),...,p^1_{im}(x)) := {x};
     endif;
end ∀;
  ⋮
(∀1 ≤ i ≤ r_ℓ | [p^ℓ_{i1}(x),...,p^ℓ_{im}(x)] ≠ [v_1,...,v_{m_ℓ}])
     if [p^ℓ_{i1}(x),...,p^ℓ_{im_ℓ}(x)] ∈ PROJECT(m_ℓ,s̄_ℓ) then
          s̄_ℓ(p^ℓ_{i1}(x),...,p^ℓ_{im_ℓ}) :=
                    s̄_ℓ(p^ℓ_{11}(x),...,p^ℓ_{im_ℓ}(x)) + {x}; else
          s̄_ℓ(p^ℓ_{i1}(x),...,p^ℓ_{im_ℓ}) := {x};
     endif;
end ∀;
  ⋮
(∀1 ≤ i ≤ r_z)
     if [p^z_{i1}(x),...,p^z_{im_z}(x)] ∈ PROJECT(m_z,s̄_z) then
          s̄_z(p^z_{i1}(x),...,p^z_{im_z}(x)) :=
                    s̄_z(p^z_{i1}(x),...,p^z_{im_z}) + {x}; else
          s̄_z(p^z_{i1}(x),...,p^z_{im_z}(x)) := {x};
     endif;
end ∀;
     /* INSERT derivative code here for */
     /* updating C̄ with respect to  x := s + {x} */
end ∀;
```

Note that we can further optimize (42') by removing code which updates any map $\bar{s}_k$, $k = 1,\ldots,z$ which does not depend on f_ℓ. At worst, the cost of (42') will be $O(z \times \delta(\sum_{i=1}^{z} r_i))$ where δ is a bound on the cardinality of the sets $\bar{s}_k(y_1,\ldots,y_{m_k})$, $k = 1,\ldots,z$.

The preceding reformulation of Rule 2 finally puts this rule into a viable form. In part, this is due to the fact that we can lift the previous restriction that all of the parameters $v_1,\ldots,v_\ell$ of the indexed assignment $f_\ell(v_1,\ldots,v_\ell) := W$ in (42') must be region constants. Perhaps more importantly, it is a result of establishing a greater coordination between application of Rule 1 (or its extension, (8)) and 2. Since we derived Rule 2 from the more basic Rule 1, it is not surprising that in all examples presented in this thesis, whenever Rule 2's enabling condition 1 (which pertains to Rule 1) holds, so does condition 2 (which is more directly related to Rule 2). It is not unreasonable, therefore, to expect that the efficient reformulated Rule 2 can be applied in the same contexts in which we can differentiate the setformer (7) with a fast variant of Rule 1. Therefore, in the rest of this chapter, we will deal mainly with formal differentiation rules based on Rule 1.

Next, consider

$$(43)\quad \{x \in s \mid \&_{i=1}^{m} (K_i(q_1,\ldots,q_n) \in K_i'(x)) \;\&\; K_{m+1}(x,b_1,\ldots,b_t)\}$$

in which $K_1, K_1', \ldots, K_m, K_m'$, and K_{m+1} are restricted as before. Then to compensate for a redefinition $s := s \pm \Delta$ of s,

we can execute the following prederivative,

(44) $(\forall x \in \Delta,\ [b_1,\ldots,b_t] \in \text{PROJECT}(t,\bar{C}),$
$d_1 \in K_1'(x) * \mathit{DOM}\ \bar{C}\{b_1,\ldots,b_t\},\ \ldots$
$d_m \in K_m'(x) * \mathit{DOM}\ \bar{C}\{b_1,\ldots,b_t,d_1,\ldots,d_{m-1}\}\ |$
$K_{m+1}(x,b_1,\ldots,b_t))$
$\bar{C}(b_1,\ldots,b_t,d_1,\ldots,d_m) := \bar{C}(b_1,\ldots,b_t,d_1,\ldots,d_m) \pm \{x\};$
end $\forall$;

Note that the boolean subparts of (36) and (43) can be conjoined within a new setformer which can then be differentiated with respect to $s := s \pm \Delta$ by using rules (37), (37') and (44) together in an obvious way. However, replacing any of these conjunctions with a disjunction will usually prevent any profitable formal differentiation.

Two more examples worth considering are

(45) $C_1 = \{x \in s \mid ([+: 1 \leq i \leq m \mid K_i(x) \notin Q_i]1) = 0$
$\& K_{m+1}(x,b_1,\ldots,b_t)\ \mathit{or}\ K_{m+2}(x,b_{t+1},\ldots,b_v)\}$

and

(45') $C_2 = \{x \in s \mid ([+: 1 \leq i \leq m \mid K_i(x) \in Q_i]1) = 0$
$\& K_{m+1}(x,b_1,\ldots,b_t)\ \mathit{or}\ K_{m+2}(x,b_{t+1},\ldots,b_v)\}$

whose continuity properties we can exploit in order to handle the setformers

$\{x \in s \mid \underset{i=1}{\overset{m}{\&}} (K_i(x) \in Q_i)\ \&\ K_{m+1}(x,b_1,\ldots,b_t)\ \mathit{or}\ K_{m+2}(x,b_{t+1},\ldots b_v)\}$

and

$\{x \in s \mid \underset{i=1}{\overset{m}{\&}} (K_i(x) \notin Q_i)\ \&\ K_{m+1}(x,b_1,\ldots,b_t)\ \mathit{or}\ K_{m+2}(x,b_{t+1},\ldots b_v)\}$

respectively equivalent to (45) and (45'). To reduce (45) and (45'), we must first reduce their compound operator subexpressions

COUNT1(x) = [+: 1 $\leq$ i $\leq$ m | $K_i(x) \notin Q_i$]1 and

COUNT2(x) = [+: 1 $\leq$ i $\leq$ m | $K_i(x) \in Q_i$]1 ,

both of which are defined on the domain s. Once this has been done, we can reduce (45) and (45') to maps $\bar{C}_1(b_1,\ldots,b_v)$ and $\bar{C}_2(b_1,\ldots,b_v)$. The prederivatives of COUNT1 and $\bar{C}_1$ corresponding to modifications $Q_j := Q_j \pm \Delta$ are given by

(46) ($\forall q \in (\Delta \bar{*} Q_j)$, $x \in \{w \in s \mid K_j(w) = q\}$,

$[b_1,\ldots,b_v] \in$ PROJECT$(v,\bar{C}_1)|K_{m+1}(x,b_1,\ldots,b_t)$

& $\neg K_{m+2}(x,b_{t+1},\ldots,b_v)$)

COUNT1(x) := COUNT1(x) $\bar{+}$ 1;

if COUNT1(x) = $\begin{matrix}0\\1\end{matrix}$ *then*

$\bar{C}_1(b_1,\ldots,b_v) := \bar{C}_1(b_1,\ldots,b_v) \pm \{x\}$;

endif;

end $\forall$;

The prederivatives of COUNT2 and $\bar{C}_2$ are

(46') ($\forall q \in (\Delta \bar{*} Q_j)$, $x \in \{w \in s \mid K_j(w) = q\}$,

$[b_1,\ldots,b_v] \in$ PROJECT$(v,\bar{C}_2)|K_{m+1}(x,b_1,\ldots,b_t)$

& $\neg K_{m+2}(x, b_{t+1},\ldots,b_v)$)

COUNT2(x) := COUNT2(x) $\pm$ 1;

if COUNT2(x) = $\begin{matrix}1\\0\end{matrix}$ *then*

$\bar{C}_2(b_1,\ldots,b_v) := \bar{C}_2(b_1,\ldots,b_v) \bar{+} \{x\}$;

endif;

end $\forall$;

Note that in both (46) and (46'), $\{w \in s \mid K_j(w) = q\}$ can be differentiated efficiently using methods presented earlier (cf. discussion of (34)).

The techniques used to handle (45) and (45') also apply to

$$(47)\quad C_3 = \{x \in s \mid ([+: 1 \leq i \leq m \mid K_i(x) \in Q_i]1) \neg= 0 \ \& \ K_{m+1}(x,b_1,\ldots,b_t) \ \textit{or} \ K_{m+2}(x,b_{t+1},\ldots,b_v)\}$$

and

$$(47')\quad C_4 = \{x \in s \mid ([+: 1 \leq i \leq m \mid K_i(x) \notin Q_i]1) \neg= 0 \ \& \ K_{m+1}(x,b_1,\ldots,b_t) \ \textit{or} \ K_{m+2}(x,b_{t+1},\ldots,b_v)\}$$

which are equivalent to

$$\{x \in s \mid (\underset{i=1}{\overset{m}{\textit{or}}} (K_i(x) \in Q_i)) \ \& \ K_{m+1}(x,b_1,\ldots,b_t) \ \textit{or} \ K_{m+2}(x,b_{t+1},\ldots,b_v)\}$$

and

$$\{x \in s \mid (\underset{i=1}{\overset{m}{\textit{or}}} (K_i(x) \notin Q_i)) \ \& \ K_{m+1}(x,b_1,\ldots,b_t) \ \textit{or} \ K_{m+2}(x,b_{t+1},\ldots,b_v)\}$$

As in the previous examples, we make use of maps COUNT2 and COUNT1 in order to reduce (47) and (47') to maps $\bar{C}_3(b_1,\ldots,b_v)$ and $\bar{C}_4(b_1,\ldots,b_v)$. The prederivative code sequences analogous to (46) and (46') for updating COUNT2, COUNT1, $\bar{C}_3$ and $\bar{C}_4$ relative to changes $Q_j := Q_j \pm \Delta$ are

(48) $(\forall q \in (\Delta \bar{*} Q_j), x \in \{w \in s \mid K_j(w) = q\},$
$[b_1,\ldots,b_v] \in PROJECT(v,\bar{C}_3) \mid K_{m+1}(x,b_1,\ldots,b_t)$
$\& \neg K_{m+2}(x,b_1,\ldots,b_t))$
$COUNT2(x) := COUNT2(x) \pm 1;$
if $COUNT2(x) = \begin{matrix}1\\0\end{matrix}$ *then* $\bar{C}_3(b_1,\ldots,b_v)$
$:= \bar{C}_3(b_1,\ldots,b_v) \pm \{x\};$
ENDIF;
end $\forall$;

for $\bar{C}_3$ and

(48') $(\forall q \in (\Delta \bar{*} Q_j), x \in \{w \in s \mid K_j(w) = q\}, [b_1,\ldots,b_v] \in PROJECT(v,\bar{C}_4)$
$\mid K_{m+1}(x,b_1,\ldots,b_t) \& \neg K_{m+2}(x,b_1,\ldots,b_t))$
$COUNT1(x) := COUNT1(x) \mp 1;$
if $COUNT1(x) = \begin{matrix}0\\1\end{matrix}$ *then* $\bar{C}_4(b_1,\ldots,b_v) :=$
$\bar{C}_4(b_1,\ldots,b_v) \mp \{x\};$
ENDIF;
END $\forall$;

for $\bar{C}_4$.

Note that whenever the formula $K_{m+1}(x,b_1,\ldots,b_t)$ $\& \neg K_{m+2}(x,b_{t+1},\ldots,b_v)$ appearing in (46), (46'), (48), and (48') can be simplified to conjunctions of terms of the form $K(x)$, $K(x) = K'(q_1,\ldots,q_m)$, or $K'(q_1,\ldots,q_m) \in T$, we can use various optimization techniques introduced earlier in this chapter to speed up the iterations within (46), (46'), (48), and (48'). Note also that set formers

$$C_5 = \{x \in s \mid K_1(x) \in Q_1 \ \&\ K_2(x,b_1,\ldots,b_t) \ \textit{or}\ K_3(x,b_{t+1},\ldots,b_v)\}$$

and

$$C_6 = \{x \in s \mid K_1(x) \notin Q_1 \ \&\ K_2(x,b_1,\ldots,b_t) \ \textit{or}\ K_3(x,b_{t+1},\ldots,b_v)\}$$

are degenerate forms of (45), (46) and (45'), (46') and can be differentiated directly without using the auxiliary maps COUNT1 and COUNT2.

As an illustrative example, consider

$$\begin{aligned}(49)\quad \{x \in s \mid (K_1(x) \neg= K_1'(q_1,\ldots,q_t)) \ \textit{or}\ (K_2(q_1,\ldots,q_t) \notin K_2'(x)) \\ \textit{or}\ K_3(x) \ \textit{or}\ K_4(x) \notin Q \ \&\ K_5(x) = K_5'(q_1,\ldots,q_t) \\ \&\ K_6(q_1,\ldots,q_t) \in K_6'(x) \ \&\ K_7(x)\}\end{aligned}$$

We can reduce (49) by storing its values in a map $\bar{C}(K_2(q_1,\ldots,q_t),\ K_6(q_1,\ldots,q_t),\ K_1'(q_1,\ldots,q_t),\ K_5'(q_1,\ldots,q_t))$ which is continuous with respect to changes $Q := Q \pm \Delta$ to Q. The following code is a fairly efficient prederivative of (49).

$$\begin{aligned}(50)\quad &(\forall q \in \Delta,\ x \in \{w \in s \mid K_4(w) = q\},\ b_1 \in K_2'(x) * \textit{DOM}\ \bar{C}, \\ &\quad b_2 \in K_6'(x) * \textit{DOM}\ \bar{C}\{b_1\} \mid \neg K_3(x) \ \&\ K_7(x) \ \& \\ &\qquad [K_1(x),K_5(x)] \in \text{PROJECT}(2,\ \bar{C}\{b_1,b_2\})) \\ &\quad \bar{C}(b_1,b_2,K_1(x),K_5(x)) := \bar{C}(b_1,b_2,K_1(x),K_5(x)) + \{x\}; \\ &\textit{END}\ \forall;\end{aligned}$$

This can be further improved by reduction of $\{w \in s \mid K_4(w) = q\}$.

We note in regard to the last several examples that quantifiers appearing in the boolean subpart of setformers can be rewritten as finite disjunctions and conjunctions. Thus, in addition to the direct methods we have already mentioned for handling quantifiers, we can also make use of

the techniques for differentiating general setformers (36), (43), (45), (45'), (47) and (47').

The techniques and concepts described in this section can be used to extend the basic continuity Rules 1 and 2 to generalized set formers involving multiple iterators, i.e.

$$(51)\quad C = \{[x_1,\ldots,x_q]: x_1 \in t_1,\ x_2 \in t_2(x_1),\ldots,x_q \in t_q(x_1,\ldots x_{q-1}) \mid K(x_1,\ldots,x_q)\}\ .$$

Here we assume that K does not contain any free occurrences of any free parameters appearing in the set expressions $t_1,\ldots,t_q$. In order to reduce the total expression (51), we must first be able to reduce all the subexpressions t_j upon which (51) depends. Let us suppose for the sake of simplicity that each expression t_j in (51) is continuous in all of its parameters other than $x_1,\ldots,x_{j-1}$ (which we will treat as 'discontinuity parameters'.) Then if the parameters on which t_j depends continuously undergo only small changes in R, we know from preceding analysis that t_j is reducible; to reduce it we store its values as a map $\bar{t}_j(x_1,\ldots,x_{j-1})$. Since for $2 \leq j \leq q$, the range of values of x_j depends on the values of $x_1,\ldots,x_{j-1}$, it is convenient to consider the set

$$D_{[x_1,\ldots,x_{i-1}]} = \{[x_1,\ldots,x_{i-1}]: x_1 \in t_1,\ldots,x_{i-1} \in t_{i-1}(x_1,\ldots x_{i-2})\}$$

as the domain of the map $\bar{t}_i(x_1,\ldots,x_{i-1})$. Whenever a parameter in which t_i is continuous changes differentially, the values of $\bar{t}_i(x_1,\ldots,x_{i-1})$ must be updated for all necessary values

of $[x_1,\ldots,x_{i-1}] \in D_{[x_1,\ldots,x_{i-1}]}$ (these sets of values are defined by rules like those discussed earlier in the present section); the actual update operations to be employed will be defined by rules like those of Section C. Moreover, since any change to $t_i(x_1,\ldots,x_{i-1})$ can also change the domains $D_{[x_1,\ldots,x_j]}$ for $j > i$, and in particular can cause them to increase, it will sometimes be necessary to calculate additional map values $\bar{t}_{i+1}(x_1,\ldots,x_i),\ldots\bar{t}_q(x_1,\ldots,x_{q-1})$ when $t_i(x_1,\ldots,x_{i-1})$ changes. Once the $t_1,\ldots t_q(x_1,\ldots,x_{q-1})$ are known to be reducible, then (51) can be reduced by replacing the subexpressions $t_1,\ldots,t_q(x_1,\ldots,x_{q-1})$ by the maps $\bar{t}_1,\ldots,\bar{t}_q(x_1,\ldots,x_{q-1})$ and evaluating the result on entrance to loop R. Then directly after any of the maps $\bar{t}_i(x_1,\ldots,x_{i-1})$ are updated within R we can insert code (derived in part from the code to update $\bar{t}_i$) which updates (51).

As an example, suppose that $t_1,\ldots,t_q(x_1,\ldots,x_{q-1})$ are all reducible to maps $\bar{t}_1,\ldots,\bar{t}_q(x_1,\ldots,x_{q-1})$, all these maps having domains as described just above, and suppose that a particular set expression t_I has the form

$$\{x \in s \mid K'(x,x_1,\ldots,x_{I-1})\}.$$

If s is changed slightly within R by an assignment $s := s - \Delta$, then the appropriate prederivative is

(52) $(\forall x_1 \in \bar{t}_1,\ldots,\forall x_{I-1} \in \bar{t}_{I-1}(x_1,\ldots,x_{I-2}))\bar{t}_I(x_1,\ldots,x_{I-1})$
$= \bar{t}_I(x_1,\ldots,x_{I-1}) - \{x \in \Delta \mid K'(x,x_1,\ldots,x_{I-1})\};;$
$C := C - \{[x_1,\ldots,x_q] : x_1 \in \bar{t}_1,\ldots,x_{I-1} \in \bar{t}_{I-1}(x_1,\ldots,x_{I-2}),$
$x_I \in \{x \in \Delta \mid K'(x,x_1,\ldots,x_{I-1}),\ x_{I+1} \in \bar{t}_{I+1}(x_1,\ldots,x_I),\ldots,$
$x_q \in \bar{t}_q(x_1,\ldots,x_{q-1}) \mid K(x_1,\ldots,x_q)\};$

However in the case of a differential modification $s:=s+\Delta$ of s, the situation is not so fortunate. In this case the necessary update operations are described by the following much more complicated nested loop which updates old values of $\bar{t}_I$ and calculates new values of $\bar{t}_{I+1},\ldots,\bar{t}_q$.

(53) $(\forall x_1 \in \bar{t}_1,\ldots,x_{I-1} \in \bar{t}_{I-1}(x_1,\ldots,x_{I-2}))\ \bar{t}_I(x_1,\ldots,x_{I-1}) :=$
$\bar{t}_I(x_1,\ldots,x_{I-1}) + \{x \in \Delta | K'(x,x_1,\ldots,x_{I-1})\}$
$(\forall x_I \in \Delta | K'(x,x_1,\ldots,x_{I-1}))\ \bar{t}_{I+1}(x_1,\ldots,x_I) :=$
$t_{I+1}(x_1,\ldots,x_I);$
$(\forall x_{I+1} \in \bar{t}_{I+1}(x_1,\ldots,x_I))\ \bar{t}_{I+2}(x_1,\ldots,x_{I+1}) :=$
$t_{I+2}(x_1,\ldots,x_{I+1}),\ \ldots,$
$(\forall x_{q-1} \in \bar{t}_{q-1}(x_1,\ldots,x_{q-2}))\ \bar{t}_q(x_1,\ldots,x_{q-1}) :=$
$t_q(x_1,\ldots,x_{q-1})$
end $\forall x_1,\ldots$, *end* $\forall x_{q-1}$;

$C := C + \{[x_1,\ldots,x_q]: x_1 \in \bar{t}_1,\ldots,x_{I-1} \in \bar{t}_{I-1}(x_1,\ldots,x_{I-2}),$
$x_I \in \{x \in \Delta | K'(x,x_1,\ldots,x_{I-1})\},\ x_{I+1} \in \bar{t}_{I+1}(x_1,\ldots,x_I),\ldots,$
$x_q \in \bar{t}_q(x_1,\ldots,x_{q-1}) \mid K(x_1,\ldots,x_q)\ ;\ .$

Despite the complexity of (53), it is not difficult to see that the work implied by the differential update calculations (52) and (53) is much less than the work necessary to perform the total calculation (51).

Thus we see that (51) is continuous in the collective differential changes to every value that the map $\bar{t}_I(x_I,\ldots x_{I-1})$ can take over its domain $D_{[x_1,\ldots,x_{I-1}]}$; and since the expression t_I is continuous in differential changes to all of its parameters except for $x_1,\ldots,x_{I-1}$, we can conclude

that (51) is continous relative to all of the continuity variables of t_I.

As a final remark concerning examples (52) and (53), we note that even when some or all of the computed set expressions t_j of (51) cannot be reduced profitably (because the space required by the maps $\bar{t}_j$ is excessive), we can sometimes reduce (51) without reducing the maps t_j. This is done by replacing (52) and/or (53) by the code

(54) $C := C \pm \{[x_1,\ldots,x_q],\ x_1 \in t_1,\ldots,x_{I-1} \in t_{I-1}(x_1,\ldots,x_{I-2}),$
$x_I \in \{x \in \Delta \mid K'(x,x_1,\ldots,x_{I+1})\ ,\ x_{I+1} \in t_{I+1}(x_1,\ldots,x_I),\ldots,$
$x_q \in t_q(x_1,\ldots,x_{q-1}) \mid K(x_1,\ldots,x_q)\};$

to be executed within L after $s := s \pm \Delta$. Note that this code will often be inferior to (52) and/or (53) for cases in which (52)/(53) can be used, since in (54) the expression t_j must be recalculated repeatedly; however, since the set $\{x \in \Delta \mid K'(x,x_1,\ldots,x_{I-1})\}$ will generally be much smaller and easier to calculate than $\{x \in s \mid K'(x,x_1,\ldots,x_{I-1}\}$, the update operation (54) may be much less burdensome than the full calculation (51).

Rule 2 generalizes to (51) much more smoothly than Rule 1. Whenever an n-ary map f (all of whose occurrences in K of (51) have at least one parameter involving a bound variable x_i) is changed within R by an indexed assignment $f(y_1,\ldots,y_n) = z$, the code required to update the value of (51) is

(55) $s_2 = \{[x_1,\ldots,x_q]: x_1 \in t_1,\ldots, x_q \in t_q(x_1,\ldots,x_{q-1})$
$\mid p_{11}(x_1,\ldots,x_q) = y_1 \;\&\ldots\&\; p_{1n}(x_1,\ldots,x_q) = y_n$ *or*
$\ldots$ *or* $p_{r1}(x_1,\ldots,x_q) = y_1 \;\&\ldots\&\; p_{rn}(x_1,\ldots,x_q) = y_n\}$;

C := C - {[x(1),...,x(q)]: x ∈ s_2|K(x(1),...,x(q))};

$f(y_1,\ldots,y_n)$:= z;

C := C + {[x(1),...,x(q)]: x ∈ s_2|K(x(1),...,x(q))};

where the notation used is like that which we have employed in the preceding discussion of Rule 2.

One last example,

(56) C = {e(x): x ∈ s | K(x)}

will illustrate reduction techniques somewhat different from those already mentioned. Expression (56) is continuous with regard to changes s := s + Δ in s, and it has the prederivative

(57) C := C + {e(x): x ∈ Δ | K(x)} .

However, Rule 1 does not generalize easily to give a prederivative for (56) with respect to changes s := s - Δ. One way of handling this difficulty is to transform (56) to the form

(58) C' = {e(x): x ∈ {w ∈ s | K(w)}}

and then reduce {w ∈ s | K(w)}. Another more powerful method begins by representing (56) as a multiset; i.e., by making use of an auxiliary map

$$COUNT(q) = \lfloor +: x \in s \mid e(x) = q \ \& \ K(x) \rfloor 1$$

we can rewrite (56) by an equivalent setformer $C_1 = \{e(x) : x \in s | COUNT(e(x)) \neq 0\}$ which is easier to handle. C_1 can be differentiated profitably with respect to the changes $s := s \pm \Delta$ using the following prederivative code,

(59) $(\forall x \in (\Delta \bar{*} s),\ e(x) \in DOM\ COUNT \mid K(x))$

$$COUNT(e(x)) = COUNT(e(x)) \pm 1;$$

END $\forall$;

$$C_1 = C_1 \pm \{e(x) : x \in \Delta \mid COUNT(e(x)) \neq 0\};$$

To deal with more general set formers than those already considered, we propose to transform them into expressions involving conveniently differentiable subexpressions on the one hand and on the other hand subexpressions which are not amenable to reduction. Chapter 3 describes a semiautomatic source to source interactive transformational system which can assist this process of program restructuring.

We have now given quite a number of illustrative examples, and can move ahead to our next chapter which will discuss implementation algorithms and general applications. However, before doing this, we shall pause momentarily and note that there exists a whole class of transitive closure algorithms which are amenable to improvement by our transformations. The main part of such transitive closure algorithms typically consist of while loops which iterate

a block of code until an existential quantifier becomes false, i.e., have the following general form:

```
          /* initialize variables */
(60)      (while ∃x ∈ s | K(x))
               <BLOCK>(x)
          END while
```

where <BLOCK> involves redefinitions $s := s \pm \Delta$ to s, indexed assignments $f(y_1,\ldots,y_n) := z$ to a map f which also has occurrences in K, and perhaps other kinds of changes to variables on which K depends (we also assume that <BLOCK> contains uses of x). The techniques presented in this chapter indicate that (60) can often be transformed by formal differentiation to yield a faster 'workset' version,

```
(61)      /* initialize variables */
          workset  := {x ∈ s | K(x)}
          (while ∃x ∈ workset)
               <BLOCK'>(x)
          END while
```

where <BLOCK'> is formed from <BLOCK> by insertion of derivative code to keep WORKSET available. Still more generally, there will exist situations in which $\{x \in s | K(x)\}$ involves discontinuity parameters $q_1,\ldots,q_n$ so that a map $\overline{\text{workset}}(q_1,\ldots,q_n)$ is necessary to store separate values of $\{x \in s | K(x)\}$ within the loop of (61).

Many sorting, parsing, graph, and general problem solving algorithms can be written in the form (60). Some of these algorithms will be found in Appendix F.

The formal differentiation techniques we have described allow algorithms to be written in a 'high style' in which complicated manipulation of worksets can be avoided at no cost since these methods can directly generate faster algorithms from these 'high style' versions. The perfection of our methods will therefore enable programmers to use powerful high level dictions to write clear high level programs which can be transformed routinely into more efficient low level versions (cf. Schl-Schl2 for a comprehensive discussion of this idea; cf. Schll, Schl2, Sch2 for particular high level dictions optimizable by formal differentiation). Among other things this will facilitate correctness proofs of programs, since, e.g., we can expect to prove undifferentiated programs of the form (60) correct more easily than their more complicated workset versions (61) (cf. Sch 10 for further elaboration of this thought).

III. An Implementation Design of Formal Differentiation of Expressions Continuous in All of Their Parameters

A. Overview and System Design

We shall now present a design for a system supporting semiautomatic formal differentiation of expressions continuous in all of their parameters. The design to be presented is a first step toward a much larger task -- i.e. an interactive program transformation system which supports formal differentiation of arbitrary expressions as well as other related program transformations such as recursion elimination. Practical implementation of such a project would require development of a large system incorporating strategies which choose between speedup goals and storage limitations. In the present chapter we concentrate exclusively on formal differentiation mechanisms which might be included in such a system, and in fact will only deal with expressions continuous in all of their parameters. This design will then be used to suggest extensions which would allow discontinuous cases to be handled.

The diagram below (Figure 1) gives an overview of our proposed system. A system user is assumed to issue instructions via an interactive terminal to a command processor which first validates his commands and then either performs them or passes them on to an appropriate supporting routine. The command processor signals successful or unsuccessful

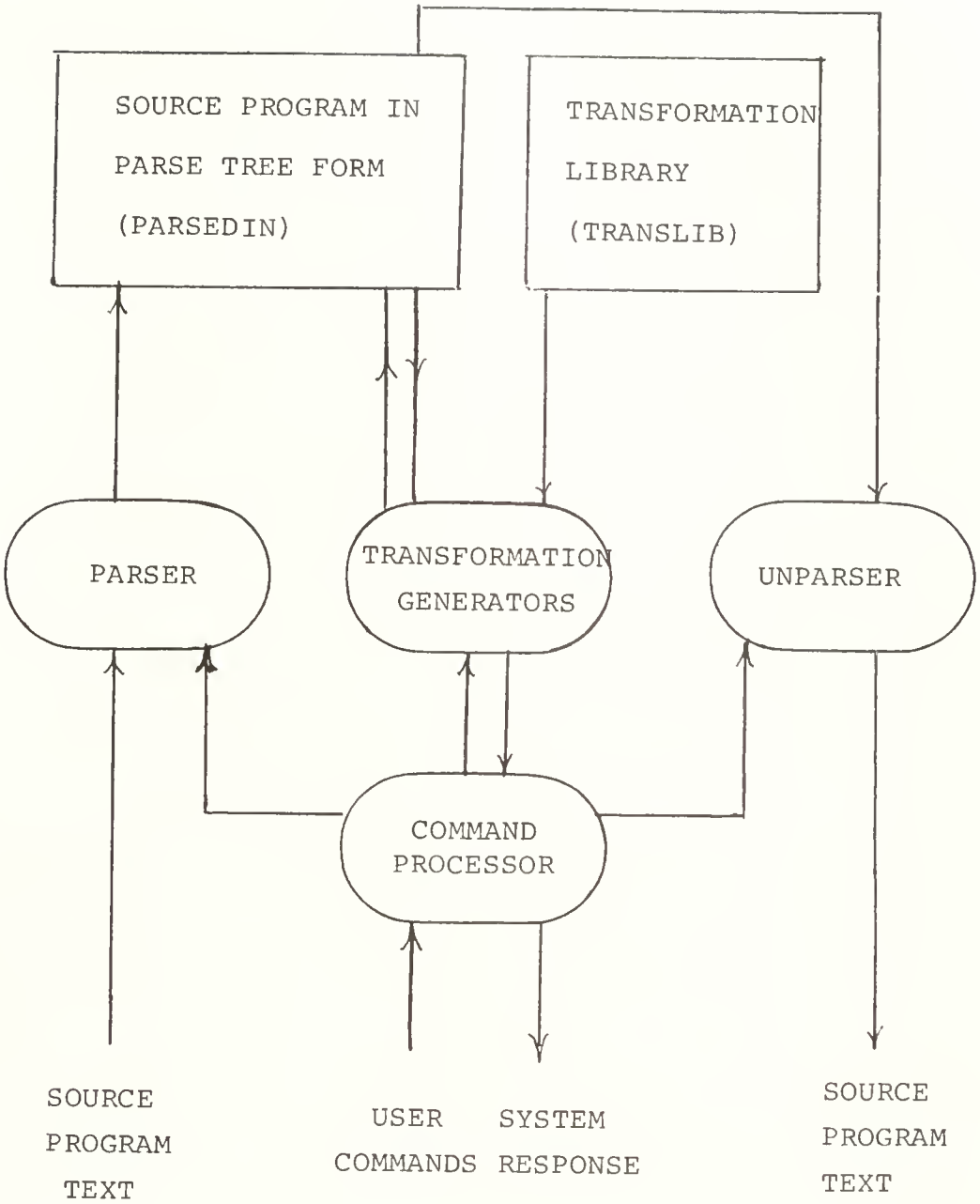


FIGURE 1. SYSTEM OVERVIEW.

completion of tasks at the user's terminal. System utilities include the following: 1. A parser reads a source program text, parses it, and outputs an annotated parse tree form of the text in a file called PARSEDIN for subsequent transformational manipulation.

2. An unparser reads PARSEDIN, unparses it, and outputs the resulting formatted program with statement numbers at the user's terminal; this utility allows the user to see the results of transformations applied to the PARSEDIN file.

3. Transformation generators which are invoked by the command processor manipulate PARSEDIN.

To perform a particular transformation T these generators find information about T in a transformation library, TRANSLIB. This information will include enabling conditions, instructions on how to manipulate PARSEDIN, and links to other transformations in TRANSLIB that can be tried next. Each generator will check that all enabling conditions are met before performing operations on the tree file. The generators may also request user intervention for difficult action or validation decisions which cannot be handled automatically.

The basic design of the transformational implementation discussed here uses ideas of Loveman [L2] and Kibler, Standish, and Neighbors [KI1,ST1,ST2]. However, the

transformations dealt with here, and formal differentiation in particular, are considerably more complex than the local syntactic transformations which largely concerned the authors just cited. Data flow, type analysis, value flow, inclusion and membership relations [Sch8], and even human intervention will sometimes be necessary to enable correct application of the transformations which the system we propose will apply.

Our proposal also differs from the previous work by our use of a variant of SETL as a source language. This choice is probably necessary to eliminate some sophisticated theorem proving (which would be called for in a lower level language) to establish enabling conditions for the powerful transformations we wish to use. The close link between language and transformations becomes clearer when we view the effort required to justify the use of a transformation as involving a temporary decompilation of a program from a language of lower level of abstraction in which the meaning of a program is scattered globally in the text into a language of higher level in which difficult semantic details are exposed only locally. The cost of this temporary decompilation can be minimized, however, if each transformation serves to translate a primitive semantic feature used in a *limited context* of a program at one language 'level' into a more efficient implementation expressed concretely at a lower level. Since some information obtainable from analysis of

more abstract versions of an algorithm will be lost at lower level versions, each application of a transformation must be chosen carefully to avoid dispersing important program facts prematurely. In other words, we are aiming for a largely top down program manipulation system.

Of course, the success of any transformational approach depends on the ease in which transformations can be seen to apply, and this is related to the difficulty of formally justifying their use.These factors all reflect the expressive power of the programming language, and the ability of this language to express everything from what Schwartz calls the most concise base 'rubble' form of an algorithm to its implementation version 'cobweb' [Schl0,Sch4].

B. SYSTEM DESCRIPTION

(i) Parser

We avoid unnecessary technical complications by describing a parser for a modified subset SUBSETL of SETL text (see Appendix A). SUBSETL lacks 'GO TO's' and labels, function and subroutine calls, I/O, and allows no side effects other than implicit assignments to the bound variables of existential and universal quantifiers. SUBSETL also contains type declarations (used for input variables) of the form <type>(<varlist>); where <type> can be INTEGER, BOOLEAN, TUPLE, SET, or MAP and <varlist> is a list of variables each separated by a comma.

A type declaration placed at a point p in a source program determines the type of all program variables found in <varlist> whose scope reaches p.

The parser first produces a parse tree version of a SUBSETL source program. A control flow graph can subsequently be worked out on the parse tree in preparation for data flow analysis and type analysis. After this analysis, the parse tree T will have been annotated with the successor and predecessor maps (FSUCC and FPRED defined on the nodes of T) of the flow graph, the map USETODEF which associates each variable use i to the set of variable definitions which can reach i, the map DEFTOUSE which associates each variable definition o to the set of variable uses reached from o, and a map TYPE mapping nodes of T representing expressions to {INTEGER, BOOLEAN, TUPLE, SET, MAP}.

(ii) UNPARSER

The purpose of the unparser is to obtain a source listing of the PARSEDIN file. The UNPARSER algorithm simply prints out the leaves of the parse tree from left to right along with the numbers of statements. Each statement begins a new line preceded by a statement number, and if a statement cannot fit on one line, it will appear on subsequent lines indented two columns to the right. In this way, the loop, block, and statement structure of the parse tree will be reflected in the indentation of the text produced by the UNPARSER.

The procedure found in Appendix E (i) represents a SETL version of UNPARSER. The parse tree consists of a set of nodes whose labels correspond to the syntactic types,

lexical types, and literals of the SUBSETL grammar. PROGRAM is the root node, TSUCC maps nodes to tuples of blank atoms representing ordered successor nodes, NUMBER maps nodes labeled <statement> to statement numbers, and LEAF is a function defined as true for leaf nodes and false otherwise.

(iii) TRANSFORMATION GENERATORS;

Our transformational approach derives from the ideas of Loveman [L2] and Standish, Kibler, Neighbors [KI1]. We describe a transformation using six of Loveman's seven properties:

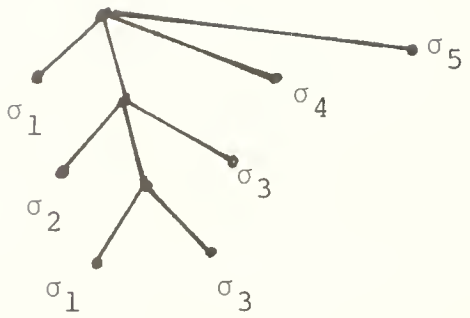
1. Name - identifies a transformation and describes the parameter format for invocation;
2. Enabling condition - a predicate which must be satisfied for the transformation to be performed;
3. Tree pattern to search for - the pattern must match a section of the tree representation of source text in order for the transformation to be performed;
4. Replacement rules - the transformational action to be performed on the tree representation of source text;
5. Changes to global functions - transformational action to be performed on such global functions as USETODEF and DEFTOUSE maps;
6. Chaining directions - these are instructions which trigger attempts to perform other transformations after the current one successfully completes.

Loveman's seventh mechanism, his 'improvement heuristic', we omit.

All the transformations of our system are encoded as 6-tuples of the form just mentioned and are stored in a transformational library, TRANSLIB. Aside from the assortment of standard program transformations catalogued in [ST2], TRANSLIB should contain a variety of set theoretic transformations important in preparing source code for formal differentiation and in cleaning up the messy code transition state that formal differentiation leaves in its wake (cf. Appendix D for a sampling of the transformations we propose to use).

Like the transformational systems of Loveman and Standish, the one described here will treat enabling conditions as *ad hoc* procedures which can interrogate global maps such as the USETODEF links,gather simple information from the annotated parse tree, or ask the user to manually validate difficult program facts. For purely local syntactic transformations, the tree pattern and replacement rules can conform to Standish's production system [KI1] as the left-hand side and right-hand side strings of productions of the form LHS => RHS. We let LHS and RHS contain pattern variables (which are denoted by capital letters), literals, and balanced pairs of parentheses. A pattern can be represented uniquely as a tree, e.g., if $\sigma_1,\sigma_2,\sigma_3,\sigma_4,\sigma_5$ are either literals or pattern variables then the pattern $(\sigma_1\ (\sigma_2\ (\sigma_1\ \sigma_3)\ \sigma_3)\ \sigma_4\ \sigma_5)$ corresponds to the following tree:

(1)



We say that a pattern (1) matches parse tree, P, if

(i) (1) has the same structure as P up to the leaves of (1).

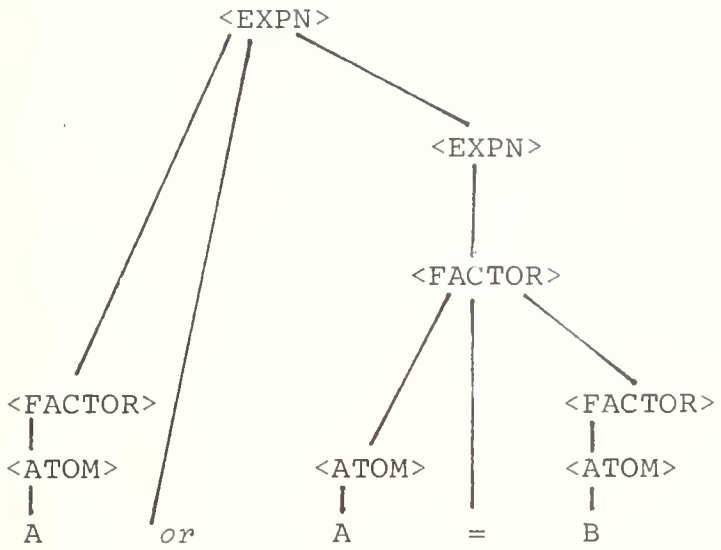
(ii) all literals of (1) match corresponding leaves of P and

(iii) all occurrences of a single pattern variable σ in (1) must match the same subtree of P.

The procedure MATCH in Appendix E (ii) is a SETL version of an algorithm which matches a pattern tree (the argument 'PATTERN') to a parse tree (the argument 'TREE'). PSUCC and TSUCC are the respective successor maps for the pattern and parse trees. The boolean function LEAF is defined as *true* for the nodes having no successors, otherwise *false*. LITERAL is a boolean valued function which returns *true* for leaf nodes which are not pattern variables of a pattern tree.

Note in connection with the foregoing that expressions enclosed in parentheses and pattern variables match subtrees. Consequently, a production, P: (X *or* X) => (X) will not match the tree (2), since the subtrees on both sides of the 'or' are not the same.

(2)



We will use the straightforward match-replacement technique just described. However, many transformations such as formal differentiation cannot be performed by a simple rewrite rule. To handle these more general procedures we let both the tree pattern and the replacement rule entry in a formal differentiation 6-tuple consist of code procedures.

We will not describe differential manipulation of global functions (e.g. data flow maps) using any systematic formalism but rather will use *ad hoc* procedures to do this. Such procedures might even have to recalculate such global functions completely.

Although chaining directions will not be used in our proposed initial system, their success as reported by Loveman [L1] and Kibler [KI1] in providing a way to automatically link several low level transformations to achieve a higher level transformation gives chaining mechanisms high

priority for future extensions to our work (cf. Appendix D, Sections VII, VIII for examples). However, the system we describe will contain lower level primitives which can support the Kibler chaining technique programmed in Appendix D.

These primitives serve to limit costly searching through the parse tree and production space by defining a tree locality and selecting transformations to try within this locality. Program localities can be defined by either a statement number (recall that each statement including compound statements are given unique sequence numbers by the parser) and/or by a pattern string. A statement number locator defines the locality to be searched as the subtree rooted to the particular statement. A pattern string locator limits the locality to be searched by matching the string against subparts of the current locality. The pattern string and matching operation in this case are somewhat less restricted than the production LHS pattern and its respective matching operation; i.e., a string locator may contain fragments of syntactic tokens and may match only part of a subtree successfully. Consequently, we expect a user of our system to be able to define a locality in his program text about as easily as he could find a portion of text by means of a general text editor.

Transformations are selected by supplying a name and proper parameters. The exact format of the commands to be supported by our system will be discussed in the following

subsection which deals with the 'command processor'.

(iv) COMMAND PROCESSOR.

The command processor (CP) interfaces between a user and the program manipulation system. One of the CP's responsibilities is to validate system commands which a user can enter from a teletype. Once validated, a command is transmitted to an appropriate utility for execution. All I/O is handled by command processor formatting and diagnostic routines. Requests for user input might originate from a utility or from the CP itself. Error diagnostics and informative messages are channelled through the CP.

The CP prompts the user to enter input by printing a prompt character > at the beginning of a line. All commands begin with a special character $ to distinguish commands from other kinds of input.

A state of the CP may be described by two components, a file table FTAB and a current tree location LOC. FTAB maps file names to tuples containing relevant file information. Each such file contains a parse tree representation of a SUBSETL program. A special file named PARSEDIN refers to the file currently being edited. LOC refers to the current locality to be searched in PARSEDIN. Initially FTAB = *nullset* and LOC = 0. A transition from one CP state to another takes place in accordance with the following list of user commands:

$PARSE,<FNAME> - CP validates <FNAME> as an accessible coded file. If validation succeeds, control passes to the parser. If parsing succeeds, PARSER outputs its parse tree of FNAME onto the file named PARSEDIN (previous contents of PARSEDIN are destroyed). FTAB(PARSEDIN) must be appropriately set and LOC refers to the root node <program> in PARSEDIN.

$UNPARSE (FNAME) - CP checks that FTAB is not empty. If this is true, it passes control to the UNPARSER utility to print out the source text at the terminal with statement numbers for the current PARSEDIN file. An optional file name parameter can be used if the user wants the source to be placed on a file instead of at the terminal. In that case a new entry for FNAME in FTAB must be made.

$L,<statement#> - This command defines a locality in terms of a particular statement. CP checks that LOC is not zero, validates the number <statement#>, and if successful it places a reference to the subtree corresponding to <statement#> in LOC. $L,±s moves the locality up or down the tree.

$P,<pattern> - This command defines a new locality within the current locality based on the first successful match between <pattern> and a sublocality according to a depth first search within the current locality. CP must first check that LOC is not 0 and that <pattern>

is well formed before the search match begins. When the first successful match occurs, LOC is set to the new node of PARSEDIN. If no successful match takes place LOC remains unchanged.

$PRINT - This command unparses the current locality if LOC $\neq$ 0.

$STOP - The program manipulation system terminates.

$SAVE,<FNAME> - The PARSEDIN file along with the current value of LOC is copied to a file named <FNAME>. CP must determine that FTAB is not empty and that a file of the name <FNAME> does not already exist before performing its functions. If all is successful CP sets FTAB(FNAME) to appropriate attribute values.

$RESUME,<FNAME> - The file PARSEDIN becomes a copy of the parsed file named <FNAME>. The old PARSEDIN file is destroyed, while the file <FNAME> remains unchanged. If the functions described just above are to be performed successfully, FTAB(FNAME) must not be undefined. LOC is also reset. Note that both the SAVE and RESUME functions provide a crude manual backtracking facility. A system user will, thus, be able to transform the same program according to different strategies at the same time. If one chain of transformations appears unfruitful, he can thus pursue an alternative strategy by RESUMing a previously saved state of the system.

$<TRANSFORMATION NAME>,<PARAMETER LIST> - This command serves to select, validate, and perform a transformation on the PARSEDIN file. CP must validate that PARSEDIN exists by checking that FTAB $\neq$ *nullset*. Then <TRANSFORMATION NAME> must match the name entry of a transformation description in TRANSLIB. The parameter list syntax must conform to the format part of the name entry. If validation succeeds, CP passes a reference to the entry in the TRANSLIB and the parameter list to the transformation generators for execution.

The above command set is an initial collection of functions to be implemented in our first system.

The following steps are necessary for system startup:

1. SUBSETL source text files are produced externally.
2. Source text files, TRANSLIB, and the program manipulation system (PMS) execute module are requested for a run.
3. When the PMS module is first executed, control passes to the CP routine which prompts the user to enter a command.
4. If FIRSTFILE is the first file that a user wants to transform, then the command $PARSE,FIRSTFILE is entered. If FIRSTFILE is successfully parsed, transformations can then be performed. Otherwise, the user must select another source file for parsing or terminate, edit FIRSTFILE, and try again.

Let us consider the following topological sort text as a first example to illustrate system functioning:

```
/* sp is a predecessor relation which defines a partial
   order on s */
t = nulltuple;
(while ∃x ∈ s | (sp{x} *s)= nullset)
t = t + [x];
s = s - {x};
end while;  /* t will be the total order */
```

which is contained in a coded file named TOPSORT. For this example, we will use two transformations from TRANSLIB, ∃FORMAT and SETEQNL (Appendix D, IX describes the collection of transformations that we assume will be available), to transform the existential quantifier of the topological sort above into a more convenient form.

During step 3 of system startup the command processor will first prompt ('>' character is used) the user to enter a command. The steps below describe a user's interaction with the system:

user command	system response
$PARSE,TOPSORT	TOPSORT is parsed and stored on PARSEDIN. LOC is set to the root node. The system then prompts the user (a prompt will always be generated after the system completes each task).
$UNPARSE	PARSEDIN is printed at the user's terminal with statement numbers and textual structuring as follows:

```
1 t = nulltuple;
2 (while ∃x ∈ s | (sp{x} * s)= nullset)
3    t = t + [x];
4    s = s - {x};
  end while;
```

$L,2 LOC is set to the statement node corresponding to the while loop. A new locality is defined.

$P,($\bar{X}$ = $\bar{Y}$) Starting from the current locality, a depth first search matches the pattern ($\bar{X}$ = $\bar{Y}$) with the expression (sp{x}*s) = *nullset*. LOC is reset to the new node.

$SETEQNL The transformation generator will immediately match the LHS pattern of the SETEQNL rule to the current locality. Consequently, (sp{x} * s) = *nullset* is replaced by ([+: u ∈ (sp{x} * s)]1) = 0 and LOC remains unchanged.

$L,2 LOC is set back to the statement corresponding to the while loop.

$∃FORMAT Starting with the current locality, a depth first search will eventually stop with a match between the LHS pattern of the ∃FORMAT rule and the existential quantifier expression in PARSEDIN. After the replacement takes place, LOC will be set to the current node (for the ∃ quantifier).

$UNPARSE The following PARSEDIN text is printed at the user's terminal:

```
1 t = nulltuple;
2 (while ∃x∈{v∈s|([+:u∈(sp{v}*s)]1) = 0})
3    t = t + [x];
4    s = s - {x};
  END WHILE;
```

$SAVE,TOP1 A new 2 record file, TOP1 is created with the contents of the current value of LOC as the first record and a copy of PARSEDIN as the second.

Next, suppose that TOP2 is a saved file containing the following parsed code (representing a lower level version of a topological sorting algorithm):

```
1     t = nulltuple;
2     (∀x ∈ s)
3           COUNT(x) := [+: y ∈ (sp{x} * s)]1;
      END ∀;
4     (∀x ∈ s)
5           succ(x) := {y ∈ s | x ∈ sp{y}};
      END ∀;
6     ZRCOUNT = {x ∈ s | COUNT(x) = 0};
7     (while ∃x ∈ ZRCOUNT)
8           t := t + [x];
9           (∀y ∈ succ(x))
10             Q := {z ∈ s | z = y};
11             ZRCOUNT := ZRCOUNT-{z ∈ Q|COUNT(z) = 0};
12             COUNT(y) := COUNT(y) - 1;
13             ZRCOUNT := ZRCOUNT + {z ∈ Q|COUNT(z) = 0};
            END ∀;
14          ZRCOUNT = ZRCOUNT - {x};
15          s = s - {x};
      END WHILE
```

The TRANSLIB entries listed in Appendix D, IX include cleanup transformations which can be used to simplify the source code above.

What follows below is a scenario of a user's interaction with the transformation system applied to the source for TOP2.

User Command	System Response
$RESUME, TOP2	The current PARSEDIN is replaced by a copy of TOP2 and LOC is restored.
$L,10 ${TO* $ *SIMP	{z∈s|z=y} is replaced by {y}*s; system prints s *INCS* {y} at the terminal and replaces {y}*s by {y}.
$L,10 $VSUBST, SCOPE=11 $VSUBST, SCOPE=13	Occurrences of Q in statements 11 and 13 are replaced by {y} after the enabling condition is validated.
$L,11 ${=NL	System prints ∀z ∈ {y}|¬COUNT(z) = 0 and replaces the setformer of statement 11 by *nullset*.
$L,11 $IDEM $USELESS $L,13 ${TOIF	ZRCOUNT:=ZRCOUNT-NULLSET becomes ZRCOUNT:=ZRCOUNT . Statements 11 is removed. *IF* COUNT(y)=0 *then* {y} *else nullset* replaces {z∈{y}|COUNT(z) = 0}
$L,13 $DISTIF1	After validating the enabling conditions the IF expression at line 13 is distributed into a conditional statement

```
                    IF  COUNT(y)=0 then ZRCOUNT:=ZRCOUNT+{y};
                      else ZRCOUNT := ZRCOUNT + nullset;
                    end if;

$IDEM
$L,-1               Locality is moved up to the statement,
                    ZRCOUNT := ZRCOUNT;

$USELESS            The useless statement is removed.
$L,13
$EMPTYELSE          Statement 13 becomes
                    IF COUNT(y)=0 then ZRCOUNT:=ZRCOUNT+{y};
                    ENDIF;

$DEADELIM,SCOPE=7   The dead code elimination routine will
                    eliminate statement 10, Q := {y} as well
                    as the last assignment statement,
                    s := s - {x};

$L,5
${to ∀              succ(x) := {y ∈ s|x ∈ sp{y}};
                    is transformed to the following loop:
                    (∀y ∈ s|x ∈ sp{y}) succ(x):=succ(x)+{y};
                    END ∀;

$L,4                ∀iterators  at statements 4 and 5 are
$∀CONC              now trivially combined into the form
                    (∀x ∈ s, y ∈ s|x ∈ sp{y})
                              succ(x) := succ(x)+{y};
                    END ∀;

$∀COMMUTE           These three commands simplify the loop
$∀SIMP*
$*COMMUTE           at statement 4 to an equivalent loop
                    (∀y ∈ s, x ∈ sp{y})
                              succ(x) := succ(x)+{y};
                    end ∀;
```

$L,4 $∀BRKUP	System prepares loop at 4 for jamming with the loop at 2.
$L,2 $JAM	The system makes sure that the jammed blocks are disjoint and carries out the transformation.
$UNPARSE,TOP	The system places source text for the current PARSEDIN reflecting all the transformations applied above on a file named TOP. We state this
$STOP	simplified version of the topological sort just below:

```
t = nultuple;
(∀w ∈ s)
   COUNT(w) := [+: y ∈ (sp{w} * s)]1;
   (∀x ∈ sp{w})
          succ(x) := succ(x) + {w};
   END ∀;
END ∀;
ZRCOUNT := {x ∈ s|COUNT(x) = 0};
(while ∃x ∈ ZRCOUNT)
   t = t + [x];
   (∀y ∈ succ(x))
     COUNT(y) := COUNT(y)-1;
     IF COUNT(y) = 0 then ZRCOUNT := ZRCOUNT - {y};
     ENDIF;
   END ∀;
   ZRCOUNT := ZRCOUNT - {x};
END while;
```

We have just presented external design specifications for a source to source transformational implementation including an extensive collection of high level transformations required to accommodate formal differentiation which we discuss separately in the next section.

C. Computing the Formal Derivative

1. Introduction.

The major implementation problems which we need to consider center on the construction of algorithms to automate formal differentiation (which we abbreviate hereafter as FD). Since FD is a basic optimization technique applicable to programs written in a variety of languages, it is useful to describe a language independent methodology from which to derive FD implementations for particular programming languages. We will develop such a methodology in this section for handling expressions continuous in all of their parameters, and we derive algorithms for implementing FD in the contexts of FORTRAN and SETL.

Aside from providing a unified approach to implementing FD in widely varying languages, the technique described here will not be bound by assumptions which limit its usefulness to expressions continuous in all parameters, and Chapter 4 will describe extensions to our framework so that it can aid in the development of FD algorithms for handling more general expressions.

We recall from Chapter 1 that a method for implementing FD must perform two principal functions:

1. Find reduction condidate expressions; and
2. Perform the FD transformation on some of these candidates.

We will describe two strategies for handling these tasks. The first strategy is a completely automatic approach that reduces all reduction candidate expressions. This approach relates to FORTRAN level reduction in strength algorithms found in [Al,Cl,C2,Kl,Schl]. In the second approach, reduction candidates are determined automatically, but selection of candidates for differentiation is done interactively from a terminal.

We propose this second approach for an initial SETL level FD implementation design to be integrated with the transformational system described in the preceding sections of this chapter. In this section we shall describe an FD design for SETL which is capable of applying many of the transformations discussed in Chapter II, C.

An important human factors goal of our design is to minimize and localize the changes made in the source code due to application of our transformations. In particular, wholesale introduction of temporary variables to hold subexpression values, which is allowable for optimizers which only transform an intermediate text, can easily make source level code unreadable. The algorithms we use to implement FD are sensitive to this concern.

2. Automatic Approach

FD is a kind of program loop optimization which generalizes code motion and which works on a single program region at a time. The program regions we will use for FD are the 'natural' loops (described in AU1) which are defined uniquely by the dominator relation on a flow graph G and by the set of back edges in G. These loops partition G into single entry strongly connected regions. For each such loop L we insert a new prologue node at a place p in the parse tree T determined by the following conditions:

i. $p \notin L$;

ii. Control flow must pass through p before entering L;

iii. The single statement succeeding p in the control flow graph on T is the entry to L.

All code pushed out of L by FD will be moved to the prologue.

To handle the two tasks of finding and reducing differentiable expressions, we proceed roughly as follows:

1. First we find an initial set $Cands_0$ of differentiable expressions and let i := 0.

2. Then we iterate steps 3 and 4 until $Cands_i = \emptyset$.

3. Remove a candidate expression from $Cands_i$ and differentiate it.

4. Let i := i+1 and include in $Cands_i$ expressions found in $Cands_{i-1}$ plus any new reduction candidates which may result from step 3.

2.1 Finding Reduction Candidates

Our automatic approach to FD demands that all differentiable expressions in L should be reduced in an order consistent with the following rule: an expression e cannot be reduced until all differentiable subexpressions of e are first reduced. Thus, the initial set $Cands_0$ of reduction candidates will include only those differentiable expressions which do not have reducible subexpressions. Our method of constructing this set looks for all expressions e in L matched by elementary expression forms found in a collection F of such forms, and that also depend only on particular combinations of region constants and 'induction' variables.

Each elementary form $f(x_1,...,x_n)$ is a pattern tree involving pattern variables $x_1,...,x_n$ and literal symbols. We represent a pattern tree by a set N of nodes (implemented as blank atoms) and a map Psucc associating each node n with a tuple Psucc(n) of successor nodes. We also make use of a map Plabel which is partially defined on N and which associates a node n with either the name of a pattern variable or a literal value (e.g., a constant or operator symbol). Patterns are used to match subtrees representing reducible expressions within the parse tree for the loop L. In SETL our implementation for parse trees is similar to our pattern tree implementation; i.e., Tsucc is the corresponding successor map in a parse tree and Label associates a parse tree node n with a literal value such as

variable name, constant, or other token value (cf. Appendix E(ii) for more details on pattern and parse tree representation).

We say that a basic form f matches a parsed expression rooted in r if the tree structures of f and r match down to the leaves of f and if the literals of f and r match identically. In SETL this is expressed more simply by the test Match(r,f,Pfunc) where match is the boolean valued SETL function given in Appendix E(ii) and described in Section B of the present chapter. We let Text(r) denote the text expression for the parse tree r, and say that Text(r) is a potential reduction candidate expression if the predicate ∃f ∈ F|Match(r,f,Pfunc) holds. Finally, we note that when Match succeeds, its parameter Pfunc will be defined as a map associating each pattern variable x of f with the root of a matched subtree Pfunc(x) of r. If we abbreviate Text(Pfunc(x)) by $\bar{x}$, then the initial set $Cands_0$ of *elementary* reduction candidate expressions consists of all expressions Text(r), r ∈ L, satisfying the following two conditions:

1. ∃f ∈ F | Match(r,f,Pfunc)
2. If $x_1,\ldots,x_n$ are all the pattern variables of f then for i = 1,...,n, $\bar{x}_i$ is either a region constant expression or an induction variable.

We categorize induction variables according to the kinds of modifications they undergo in L. By constructing

such categories appropriately, we can decide whether within L an expression is continuous relative to all modifications to variables on which it depends. To see how this is done, it is convenient to regard each elementary form $f(x_1,...,x_n)$ in F as an elementary expression $E = f(x_1,...,x_n)$ in the variables $x_1,...,x_n$. We can then specify the kinds of modifications to $x_1,...,x_n$ relative to which f varies continuously and can describe associated update corrections to E by entries in a table of derivatives D. More specifically, if E is continuous with respect to a change $x_I := g(y_1,...,y_m)$ and if the pre and post derivatives to E are

E := preD($y_1,...,y_m,x_1,...,x_n$,E) and

E := postD($y_1,...,y_m,x_1,...,x_n$,E), then the set $D(x_I,f)$ will contain the triple

(1) [x_I := $g(y_1,...,y_m)$, E := preD($y_1,...,y_m,x_1,...,x_n$,E),
E := postD($y_1,...,y_m,x_1,...,x_n$,E)]

Once the set D of triples (1) is formed we can regard the three components of (1) as patterns for use in constructing induction variables and also in generating derivative code. Also, we can use our tree pattern matching routine Match (given in Appendix E) to match actual modifications in variables within L by parameter definition patterns stored as the first component of triples (1) contained in D. Additionally, we ease the task of recognizing redefinitions

to variables on which reduction candidates depend by assuming that variable types are made available at compile time and that all occurrences of a variable of the same name are bound to the same data structure.

In general, we must allow a different set of induction variables for every component of every elementary expression in F. Given a variable v found in L and an elementary form $f(x_1,\ldots,x_n)$, v belongs in the i-th induction set for f, which we denote $IV(x_i,f)$, iff the following two conditions hold:

1. All definitions of v in the loop L in which FD is applied match parameter definition patterns in $D(x_i,f)$.
2. For each such definition pattern the corresponding derivatives must consist of easy calculations relative to f.

We indicate costly subparts of derivatives by underlining them in the D tables of Appendix C, and require that such subparts be reducible. Thus, each subpart must match an elementary expression g found in F, and must depend on induction variables of g.

A procedure to find induction variables satisfying the two conditions above can be based largely on the F and D tables. Of course, the F and D tables must reflect some appropriate, even if loose, idea of the relative cost of operations. This informal measure of cost guides us in determining what elementary expressions to include in F,

what formal derivative rules to place in D, and also what subexpressions of these derivatives must also be reduced in order to make FD profitable.

Before describing our fully automatic reduction algorithm, we note that condition 2 above, which governs the construction of sets of induction variables, involves a recursive step taken whenever one application of an FD transformation leads to a chain of successive transformations. To prevent the possibility of infinite chains of steps of this sort, we admit only a bounded number of new differentiable expressions introduced as part of derivative code. In this connection we use the following general heuristic: never reduce an expression which has already been reduced.

2.2 Reduction Algorithm

We shall now give additional details concerning the procedures used to detect induction variables and reduction candidate expressions. These procedures employ patterns stored in our tables F and D to match expressions and statements in a loop L. Derivative patterns stored in the D table will be used as macros which generate actual derivative code which is inserted into L. Expansion of these syntax macros can involve simple text substitution in which pattern variables function as substitutable parameters. Together, substitution and matching allow us to handle a general family of differentiable expressions formed by

composition from elementary expressions.

In SETL, the function subprogram Expand(f,Pfunc) (cf., Appendix E(iii)) implements macro expansion. The two parameters of Expand are a pattern tree f and a map Pfunc which associates each pattern variables x of f with a parse tree Pfunc(x). Expand(f,Pfunc) will return the root of a new parse tree which results from replacing each pattern variable x of f by Pfunc(x).

To facilitate our discussion of the pattern matching and macro expression mechanisms used by the FD algorithms to be presented, it is convenient to make use of a few additional notational devices. If a pattern f matches a tree t so that Match(t,f,Pfunc) holds, then we use the symbol $\bar{f}$ as an abbreviation for Text(t). Note that the parameter Pfunc will be defined after successful matching, and that subsequent execution of Expand(f,Pfunc) will produce a copy of the tree t originally matched by f. We will sometimes use the term $f(x_1,\ldots,x_n)$ to denote the pattern f along with all of its pattern variables $x_1,\ldots,x_n$. In this case, we use the term $\bar{f}(\bar{x}_1,\ldots,\bar{x}_n)$ as an abbreviation for Text(t), given that Match(t,$f(x_1,\ldots,x_n)$,Pfunc) holds and that $\bar{x}_i$ = Pfunc(x_i), i = 1,...,n. If for i = 1,...,n t_i is a tree and y_i = Text(t_i), then we also use the term $\bar{f}(y_1,\ldots,y_n)$ to express the same thing as Text(Expand(f,Pfunc)), where Pfunc(x_i) = t_i , i=1,...,n.

The notation just described allows us to describe FD

transformations at both implementation and abstract levels. Our implementation requires all transformations to manipulate the parse tree form of source code. However, for clarity we will often prefer to discuss FD and other transformations more abstractly in terms of changes in source code independently of the underlying parse tree.

We will make use of the preceding notation to sketch the logic of the automatic FD procedure given below. An important characteristic of this procedure is that it only reduces elementary differentiable expressions. However, nonelementary differentiable expressions become elementary after reduction of all of their subexpressions. Thus, all differentiable expressions in L will eventually be reduced.

Algorithm 1-2.

Input: a derivative table D, a set of elementary forms F, a parse tree L of the optimization loop, and a map Defs which associates each variable name v occurring in L with the set Defs(v) of nodes in L corresponding to statements which can modify the value of v.

Output: a new optimized loop L' and its prologue code block.

1. Find the set RC of nodes in L corresponding to region constant expressions of L.
2. Compute initial sets IV(x,f) of induction variables for every elementary form $f \in F$ and each pattern variable x of f.

3. Initialize Prologue to an empty code block.

4. While $\exists$ a node $t \in L$ and an elementary form $f(x_1,\ldots,x_n) \in F$ such that

   (1) Match(t,f,Pfunc) and

   (2) for i = 1,...,n either $Pfunc(x_i) \in RC$ *or* $\bar{x}_i \in IV(x_i,f)$

   perform steps 5, 6 and 7.

5. Generate a unique variable $v_{\bar{f}}$ for keeping the matched expression $\bar{f}$ available in L, and insert an assignment $v_{\bar{f}} = \bar{f}$ at the end of the Prologue block.

6. For each expression $\bar{x}_i$, i =1,...,n such that $\bar{x}_i \in IV(x_i,f)$, and for each program point $p \in Defs(\bar{x})$ at which $\bar{x}$ undergoes a change $\bar{x} = \Delta_{\bar{x}}$, insert appropriate derivative code which keeps $v_{\bar{f}}$ available in L. This derivative code can be generated by first finding the unique triple [mod,preD,postD] belonging to $D(x_i,f)$ in which Match(p,mod,Qfunc) holds. Next, to prepare for macro expansion we must produce a new pattern variable map Sfunc (which can be formed from Pfunc and Qfunc) which maps pattern variables found in preD and postD into appropriate trees. Finally, we expand the pre and post derivative patterns preD and postD by executing Expand(preD,Sfunc) and Expand(postD,Sfunc), and insert the resulting code immediately before and after p.

7. Within L replace all occurrences of $\bar{f}$ by $v_{\bar{f}}$. Also, within the derivative code generated in step 6 substitute the variable v_e for any expression e which has

already been reduced. Finally, make appropriate additions to the set RC of region constants and to the induction sets IV.

Steps 1-5 and 7 above are fairly straightforward, but step 6 requires further explanation. We note, first of all, that as a consequence of the way our algorithm chooses expressions to reduce, each expression chosen will be elementary; i.e., if an expression pattern $f(x_1,\ldots,x_n)$ arising in step 4 is the elementary form in F matching the subtree t in L, we will know that for $i = 1,\ldots,n$, $\bar{x}_i$ is either an induction variable belonging to $IV(x_i,f)$, or a region constant expression. Hence, in computing the formal derivative of $v_{\bar{f}} = \bar{f}(\bar{x}_1,\ldots,\bar{x}_n)$ relative to the change in a variable $\bar{x}_i$, we only need to consider two cases.

1. In the simplest case the variable $\bar{x}_i$ occurs only once in $\bar{f}$. In this case, at each point p within L where $\bar{x}_i$ undergoes a change $\bar{x}_i = \Delta_{\bar{x}_i}$, we compute pre and post derivatives for $v_{\bar{f}}$ by matching $\bar{p}$ to a parameter change pattern $x_i = g(x_1,\ldots,x_n,\ldots,x_m)$ in $D(x_i,f)$ where $x_1,\ldots,x_n$ must match the same objects $\bar{x}_1,\ldots,\bar{x}_n$ in L as are matched by the corresponding pattern variables in $f(x_1,\ldots,x_n)$. Remaining pattern variables $x_{n+1},\ldots,x_m$ may match arbitrary subtrees of p. Recall here that the matched parameter change pattern is the first component of a triple

$[x_i = g(x_1,\ldots,x_m), preD(x_1,\ldots,x_m,E), postD(x_1,\ldots,x_m,E)]$

which we use to determine the derivative code relative to the change $\bar{x}_i = \Delta_{\bar{x}_i}$. Specifically, we insert code produced by expansion (involving simple text substitution)

(2) $\overline{preD}(\bar{x}_1,\ldots,\bar{x}_m,v_{\bar{f}})$ and $\overline{postD}(\bar{x}_1,\ldots,\bar{x}_m,v_{\bar{f}})$,

immediately before and after p.

Note also that the actual expansion implied by (2) can be empty.

2. A second more complicated case arises when more than one pattern variable of $f(x_1,\ldots,x_n)$ matches the same variable of $\bar{f}$. Consider an expression

(3) $$v_{\bar{f}} = \bar{f}(x,\ldots,x,\bar{x}_{j+1},\ldots,\bar{x}_n)$$

matched by $f(x_1,\ldots,x_n)$ in which for $i = 1,\ldots,j$, $x \in IV(x_i,f)$. Suppose also that for $i = j+1,\ldots,n$, either $\bar{x}_i \in IV(x_i,f)$ and $\bar{x}_i$ is different from x or $\bar{x}_i$ is a region constant expression. Then when $v_{\bar{f}}$ is available just prior to a definition $x = \Delta_x$ which spoils $v_{\bar{f}}$, we can keep $v_{\bar{f}}$ available after this change by executing the following derivative code:

(4) $x_{OLD} = x$ /* copy the old value of x */

$x = \Delta_x$ /* Δ_x is the change in x within L */

$\overline{preD_1}(x_{OLD},\ldots,x_{OLD},\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$

$\overline{postD_1}(x,x_{OLD},\ldots,x_{OLD},\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$

$\vdots$

$\overline{preD_j}(x,\ldots,x,x_{OLD},\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$

$\overline{postD_j}(x,\ldots,x,\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$

where for i = 1,...,j the macros $preD_i$, $postD_i$ are the second and third components of a unique triple (contained in $D(x_i,f)$) whose first component is a pattern matching the assignment $x = \Delta_x$.

In deriving (4) we use the formal device of replacing the j occurrences of x in (3) by uniquely renamed new variables $\bar{x}_1,\ldots,\bar{x}_m$ all having the same value of x just before the change $x = \Delta_x$, and modified by

(5)
$$\begin{aligned} \bar{x}_1 &= \Delta_{\bar{x}_1} \\ &\vdots \\ \bar{x}_j &= \Delta_{\bar{x}_j} \end{aligned}$$

just afterwards. This calls for j applications of the case 1 differentiation rule to the expression $\bar{f}(\bar{x}_1,\ldots,\bar{x}_n)$ formed from (3) by substitution. Observation shows that for each associated pre and post derivative code fragment $\overline{preD_i}$ and $\overline{postD_i}$, i = 1,...,j, all occurrences of $\bar{x}_1,\ldots,\bar{x}_{i-1}$ in $\overline{pre\ D_i}$ and occurrences of $\bar{x}_1,\ldots,\bar{x}_i$ in

$\overline{\text{post } D_i}$ will have the same value as the changed value of x; these occurrences can be replaced by occurrences of the variable x. We also note that all occurrences of $\bar{x}_i,\ldots,\bar{x}_j$ in $\overline{\text{preD}_i}$ and occurrences of $\bar{x}_{i+1},\ldots,\bar{x}_j$ in $\overline{\text{postD}_i}$ have the same value as the initial value of x; such occurrences may therefore be replaced by occurrences of the variable x_{OLD}, i.e., by a copy of the initial value of x. The code sequence (4) then results by elimination of all dead assignments to the renamed variables $\bar{x}_1,\ldots,\bar{x}_j$.

Note finally that the copy operation $x_{OLD} = x$ can sometimes be eliminated profitably by using the following approach: Find the smallest number L between 1 and j+1 such that none of the code fragments $\overline{\text{preD}_i}$ and $\overline{\text{postD}_i}$ $i = L,\ldots,j$ in (4) refer to x_{OLD}. Replace all occurrences of x_{OLD} and x in $\overline{\text{preD}_i}$ and $\overline{\text{post } D_i}$, $i = 1,\ldots,L-1$ by x and Δ_x respectively. Replace all occurrences of x in $\overline{\text{preD}_i}$ and $\overline{\text{postD}_i}$, $i = L,\ldots,j$ by x. These substitutions allow us to replace (4) by the following code:

(6)
$$\overline{\text{preD}_1}(x,\ldots,x,\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$$
$$\overline{\text{postD}_1}(\Delta_x,x,\ldots,x,\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$$
$$\vdots$$
$$\overline{\text{preD}_{L-1}}(\underbrace{\Delta_x,\ldots,\Delta_x}_{L-2},x,\ldots,x,\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$$
$$\overline{\text{postD}_{L-1}}(\underbrace{\Delta_x,\ldots,\Delta_x}_{L-1},x,\ldots,x,\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$$
$$x = \Delta_x$$

$$\overline{\text{preD}}_L(x,\ldots,x,\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$$
$$\overline{\text{postD}}_L(x,\ldots,x,\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$$
$$\vdots$$
$$\overline{\text{preD}}_j(x,\ldots,x,\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$$
$$\overline{\text{postD}}_j(x,\ldots,x,\bar{x}_{j+1},\ldots,\bar{x}_m,v_{\bar{f}})$$

Since the code (6) is defined by the way in which we order occurrences of x in (3), we can change the ordering (5) so as to generate a minimal number of calculations Δ_x. When such calculations Δ_x can be eliminated entirely or when they can be effectively eliminated by means of cleanup transformations (cf. Appendix D) then (6) will usually represent an improvement over (4). Note also that copying of nonelementary variables (e.g. array, structure or set valued variables) is likely to be expensive, so that an improved version of the update code (6) may be a necessary precondition for deciding to reduce (3) at all.

2.3 Automatic FD for FORTRAN and SETL

We can use the preceding framework to derive an FD implementation design for a given programming language Q by constructing F and D tables, by giving a procedure for finding induction variables, and by bounding the number of expressions reduced by Algorithm 1-2 for Q.

As a first example of this observation, consider the case of FORTRAN. Our goal in FORTRAN level FD is to reduce costly exponentiations to less costly multiplications, and to replace division and multiplication operations by inexpensive additions and subtractions. The Fortran F and D tables shown in Appendix C, i reflect this aim and also reflect our assumptions about the relative cost of these operations. Note that subexpressions underlined in the D table represent costly subparts of derivatives which must be further reduced to make these derivatives profitable.

If we examine the F and D tables for FORTRAN closely, we can see that use of a separate induction variable procedure for each parameter of every elementary expression is easily avoided. The Fortran table F contains three elementary forms, but the basic parameter definition patterns contained in D for each of these forms are the same. To decide whether a variable v fits these basic definition patterns we can simply evaluate the predicate

Redefl(v) ↔ each definition to v in L is matched by any of the forms $v = \pm x3$, $v = -x3 \pm x4$, or $v = x3 \pm x4$

where $\overline{x3}$ and $\overline{x4}$ can be any variable or constant.

Observe, however, that for v to be a FORTRAN induction variable, we require that the code produced by expansion of every underlined subpart of the derivative patterns (in D) associated with each definition to v in L must be an elementary reducible expression; and the variables on which this reducible expression depends must all be induction variables. In the case of FORTRAN only one set of induction variables needs to be defined for all entries in F and for each component of each entry. To test whether a variable v is in this set, we can use the predicate defined as follows:

Fortind(v) ↔ Redefl(v) &
∀definitions to v within L,
if the definition is matched by the forms
$v = \pm x3$ *then*
Fortind($\overline{x3}$) *else*
if the definition is matched by either
$v = -x3 \pm x4$ *or* $v = x3 \pm x4$ *then*
Fortind($\overline{x3}$) & Fortind($\overline{x4}$) *else*
False.

In order to make Fortind a terminating algorithm, we must substitute the value *True* for any recursive call to Fortind which passes an argument which has been previously passed. Note finally that Fortind can be used to test variables matching both parameters of x1 * x3, the parameter x1 in x1/x2, and x2 in x1**x2. Since the D table does not specify any parameters changes for x2 in x1/x2 and x1 in x1**x2, only region constants are permitted for these parameters.

To show termination of Algorithm 1-2 we use the fact that in any program there can exist only a finite number of induction variables and region constants. Thus, in reducing any binary operation, only a finite number of other different costly binary operations can be generated and subsequently reduced.

In contrast to FORTRAN, the task of recognizing the relative cost of set theoretic operations, necessary for constructing SETL F and D tables, is not so simple. The main difficulty is in statically estimating the relative sizes of sets (which helps to deduce whether an expression iterates over a large or small set). One possible way of making this estimation automatic might involve a special type of global analysis, specifically by incorporating the property 'this data object is a small set' in a monotone framework to which Kildall's technique [KI2] applies. The approach taken in this thesis, however, is much simpler

since we only need to determine the relative sizes of sets Δ and x in the context of an assignment $x := x \pm \Delta$ executed repeatedly in a program loop L. As shown by the case studies in Chapter 4 and Appendix F, in such contexts it is highly likely that Δ will be small relative to x.

F and D tables for SETL are shown in Appendix C, ii. Every derivative shown in D is considered profitable *a priori* except for subcase 5, Rule 2 in which we require that the set s_0 must be reduced to make FD worthwhile.

A straightforward procedure for finding induction variables for SETL is also available. The F and D tables shown in Appendix C, ii suggest that this algorithm should work with six different sets of induction variables, and that no recursion is required. We define these six sets as follows:

1. IV_1 = {all set valued variables x all of whose redefinitions in L are of the form $x := x + \Delta$}
2. IV_2 = {all set valued variables x which are only modified in L according to the rule $x := x- \Delta$}
3. IV_3 = {all map variables f affected only by indexed assignments $f(y_1,\ldots,y_n) := z$ in L where $y_1,\ldots,y_n$ are region constants}
4. IV_4 = {all integer variables x which change only by $x := x \pm \Delta$ in L}
5. IV_5 = {all tuple variables x that only vary in L in the the following ways $x := x + x3$, $x(x3) := x4$, and $x(x3:x4) := x5$}

6. IV_6 = {all tuple variables x which only undergo modifications of the form x = x3 + x in L}.

Any elementary expression of the forms 1, 2, 3, 4, 11, and 12 of the F table which depend on variables in $IV_1 \cup IV_2$ or on region constants are considered to be reduction candidates. Differentiable expressions 9 must depend on region constants and arguments $\overline{x1}$ and $\overline{x2}$ which are found in IV_4 and the set RC of region constants. For expressions of the forms 6, 8, and 10, we require that each non loop invariant argument belong to IV_1. Subclass 7 of 6, however, may also involve variables in IV_2. Induction variables $\overline{x1}$ and $\overline{x2}$ of expression class 13 must belong to $IV_6 \cup RC$ and $IV_5 \cup RC$ respectively. Set former expressions matching the basic reducible form number 5 will be considered differentiable if $\overline{x1} \in (IV_1 \cup IV_2 \cup RC)$ and if all map variables (which match to special patterns denoted by f_K in the D table) occurring in the boolean subexpression $\bar{K}$ belongs to $IV_3 \cup RC$. For expressions $\{\bar{x} \in \bar{s} | \bar{K}(\bar{x})\}$ of form number 5, we further stipulate that all occurrences of map induction variables within $\bar{K}$ must head map retrieval terms involving the bound variable $\bar{x}$ of the set former.

To show termination of Algorithm 1-2 for SETL, we can give a bound on the number of differentiable expressions introduced by FD based on the number of indexed assignments to map induction variables and to the f-depth of auxiliary sets generated as derivative code from basic form #5 of Appendix C ii (cf. Chapter 2 for further discussion of Rule 2).

3. A Semiautomatic Approach

In SETL, expressions can depend on large sets and maps so that a strategy for FD which seeks to differentiate an expression e only after first reducing all subexpressions of e may be prohibitively expensive in space usage. Therefore, it may be useful to consider an alternative strategy which trades off speed for space by differentiating expressions with unreduced subexpressions.

As an example of this, consider

(7) $$c = \{x \in (s + t) \mid K(x)\}.$$

If we differentiate (7) without first reducing the union operation s + t, we save the space which would be required for storing c' = s + t if c' were also reduced. Note also that by avoiding reduction of c' we can even gain speed, since the prederivatives of c relative to the changes $s := s \pm \Delta$ are

(8) $$c = c + \{x \in \Delta \mid K(x)\} \quad \text{and}$$
$$c = c - \{x \in \Delta \mid x \notin t \ \& \ K(x)\}$$

respectively.

However, as yet we lack an automatic strategy which deals with space/time tradeoffs. Hence we propose an interactive system in which expressions are manually selected for reduction, one at a time. Of course, before

an expression can be selected it must be marked 'reducible' by a routine described below, which we will call algorithm 1.

To select a qualified expression for reduction, a user then issues the command,

(9) $FD, LOOP#, NAME = EXP

from his terminal; this directs the FD transformation generators to differentiate the expression EXP, and to keep its value available within a uniquely named variable NAME throughout a loop identified by a statement number, LOOP#. Our proposed system will automatically validate this command before actually performing FD by use of a procedure we will refer to as Algorithm 2.

Although our description of Algorithms 1 and 2 will be language independent, these algorithms should only be implemented for languages in which space utilization can be of overriding concern. Since the cost of extra storage required by the Fortran level FD is low, Algorithm 1-2 is preferable in the FORTRAN context. However, we expect that the semiautomatic FD approach we are about to describe can be tailored effectively to very high level languages such as SETL, SNOBOL, and APL.

3.1 Algorithm 1.

Once induction sets and the set of region constants RC are computed, we can find all nodes n in the parse tree of a program loop L which correspond to differentiable expressions. The procedure we will sketch for doing this is bottom up in that inner expressions are handled before outer expressions; i.e., the algorithm starts with the leaves of a parse tree representation of an expression proceeds to predecessors and decides reducibility along the way. To decide reducibility, Algorithm 1 uses the following criteria. An expression e in L is reducible if e is an elementary reducible expression (cf. the definition of $CANDS_0$ in Section 2.1); e will also be considered reducible if it is matched by some $f \in F$ and if each subexpression $\bar{x}$ (of e) matched by a pattern variable x of f is either a region constant expression, a member of the induction variable set IV(x,f), or an *induction expression* for f and x; i.e., $\bar{x}$ is an induction expression for f and x if $\bar{x}$ is reducible, and once reduced with its value kept available in a variable t, t would belong in IV(x,f).

The overall logic of Algorithm 1 is as follows:

1. For each leaf ℓ in L, if ℓ corresponds to a region constant of L mark it 'good'; otherwise, if ℓ is contained in a subtree e matched by an elementary form f in F in which ℓ is matched by some pattern variable x in f and

if $\bar{x} \in IV(x,f)$ then mark ℓ 'good'.

2. Repeat step 3 until no more nodes in L can be marked.

3. For each unmarked node $n \in L$ and for each form $f \in F$, if the two conditions

   (1) Match(n,f,Pfunc) and

   (2) For all pattern variables x in f the node Pfunc(x) is marked 'good'

both hold, we will mark n 'reducible' and associate n with f. We will also mark n good if it represents a region constant or an induction subexpression $\bar{x}$ of an outer expression e matched by a basic form $f' \in F$. We can determine whether $\bar{x}$ is an induction expression for x and f' in the following way. Let W be the set of all derivative patterns in D for the basic form f such that costly subparts within these patterns are considered single pattern variables. (Recall that in the D tables of Appendix C such subparts are underlined.) If the pattern variable E denoting the value of f could be defined by any of the definitions to E found in W and still qualify as a member of IV(x,f') then $\bar{x}$ is an induction expression for x and f'.

3.2 Specializations

In the very familiar FORTRAN case all derivative patterns in the D table (cf. Appendix C(i)) except for the last two entries of D exactly match parameter change patterns in D. Thus every reducible expression 1 (products) and 2 (quotients) of F and each reducible exponentiation which only depends on the parameter changes $x_2 = \pm x3$ (where x3 must also be restricted) may be considered as inductive subexpressions of all three kinds of expressions which occur in the F table.

The following simplified FORTRAN marking algorithm exploits this fact. In it we use the predicate,

Fortindl(v) $\leftrightarrow$ all definitions to v in L
are of the form $v = \pm x3$ and
Fortindl($\overline{x3}$) holds;

This works in conjunction with the induction variable predicate Fortind (discussed in subsection 2.3) to find induction expressions. We also make use of a map Mark, partially defined on L, to indicate induction and reducible expressions.

Algorithm lFORT

1. Initialize the map Mark to *nullset*.
   Find the set RC of nodes in L corresponding to region constant expressions of L.
2. For each leaf $\ell \in L$ such that $\ell \notin RC$,
   if Fortindl(Text(ℓ)) *then*
   assign 'INDl' to Mark(ℓ); otherwise,
   if Fortind(Text(ℓ)) *then*
   assign 'IND2' to Mark(ℓ);
3. Repeat step 4 exhaustively.
4. Separate all nonterminal nodes n in L for which
   i. Mark(n) is undefined;
   ii. $n \notin RC$;
   iii. For each successor node $y \in$ Tsucc(n),
   either $y \in RC$ or Mark(y) is defined;
   iv. $f \in F$ such that Match(n,f,Pfunc) holds;
   (note that F is defined in Appendix C (i))
   into one of the following three cases:
   a) If xl * x2 matches n and $\overline{xl}$ (respectively $\overline{x2}$) is a region constant expression, assign the value of Mark(Pfunc(x2)) (respectively Mark(Pfunc(xl))) to Mark(n); else, if Mark(Pfunc(xl)) (Mark(Pfunc(x2))) is equal to 'INDl', assign the value Mark(Pfunc(x2)) (Mark(Pfunc(xl)) to Mark(n); otherwise set Mark(n) to 'IND2'.

b) If x1/x2 matches n and $\overline{x2}$ is a region constant expression, perform the assignment
Mark(n) := Mark(Pfunc(x1)).

c) If x1 ** x2 matches n and $\overline{x1}$ is a region constant expression, perform Mark(n) := Mark(Pfunc(x2)).

After lFORT terminates, all expressions Text(n) such that Mark(n) is defined and n is a nonterminal node of L will be reducible.

A variant of algorithm 1 adapted to SETL has more cases to consider but is no more complicated than lFORT. The F and D tables for SETL are defined in Appendix C(ii). First, we need to calculate the set of region constants RC and the induction variable sets $IV_1,\ldots,IV_6$ (cf. subsection 2.3). This allows us to detect the elementary reducible expressions. In order to pick out all the reducible expressions, we must be able to determine induction expressions. Fortunately, the primary induction expressions of interest are set union, intersection, set difference, and set former (these are forms 1,2,3,5,8 and 9 in our F table) whose derivative patterns in the D table only realize changes according to the forms $x := x \pm \Delta$. Consequently, we can treat reducible expressions of the kinds just mentioned as induction expressions that behave in much the same way as induction variables belonging to IV_1 and IV_2. This provides a way to follow the logic of Algorithm 1 (but with a few

minor adjustments to be discussed in Chapter 4) and locate the remaining nonelementary reducible expressions.

To illustrate the preceding remarks, consider the expression

(10) $c = \{x \in (T - S) \mid f(x) = q\} - Q$

occurring in a loop L to be optimized. Suppose that, within L, the set s is modified exclusively by changes of the form $S := S - \Delta$, Q varies only by set additions, $Q := Q + \Delta$, and T, f and q are all region constants. Then a SETL version of Algorithm 1 will decide that S and Q are induction variables belonging to IV_2 and IV_1 respectively. The first reducible subexpression of c to be detected will be $c_1 = T - S$. Since s only undergoes set deletions, we know that the value of c_1 can only grow by set additions. Thus, we will consider c_1 to be an induction expression belonging to IV_1. At this point the expression $c_2 = \{x \in c_1 | f(x) = q\}$ can be marked reducible. Moreover, since the subexpression c_1 belongs to IV_1, c_2 is also an induction expression belonging ito IV_1. Finally, since both c_2 and Q belong to IV_1, the expression $c = c_2 - Q$ is reducible and is an induction expression in both categories IV_1 and IV_2.

3.3 Algorithm 2

After Algorithm 1 has determined the differentiable subexpressions in a program loop L, a user will be able to select these subexpressions one at a time for reduction. Selection is made by commands of the form (9), which will cause the reduction routine Algorithm 2 to execute. The input and output specifications of Algorithm 2 are the same as those for Algorithm 1-2, and these two algorithms have rather similar logic. However, while Algorithm 1-2 ensures profit by reducing all reducible expressions in L, Algorithm 2 only attempts to reduce as few reducible subexpressions of EXP as possible (so as to conserve space) without sacrificing expected speedup.

Algorithm 2.

1. To validate the command (9), we check that LOOP# refers to a program loop L, that NAME is not a program variable which already exists, and that the expression EXP is located in L at a node n which has been marked 'reducible' by Algorithm 1.

2. Next we order the reducible subexpressions of EXP in a postorder arrangement (cf. Appendix E (iv)) as indicated by the following SETL assignment,

```
Cands := [x ∈ Postorder(n) |Marked(x)='reducible']
```

3. For each node t selected from the tuple Cands, determine the particular elementary form f in F for which Match(t,f,pfunc) holds, and perform steps 5-8.

4. Halt.

5. Generate a unique name $v_{\bar{f}}$ for the variable which will hold the value of the subexpression $\bar{f}$ in L, and insert an assignment $v_{\bar{f}} := \bar{f}$ at the end of the prologue for L.

6. For each pattern variable x in f for which $\bar{x}$ is an induction variable belonging to IV(x,f), and for each program point $p \in Defs(\bar{x})$ at which $\bar{x}$ undergoes a change $\bar{x} = \Delta_{\bar{x}}$, insert derivative code which keeps $v_{\bar{f}}$ available in L. Since the ordering of nodes in Cands and the overall strategy of our algorithm makes $\bar{f}$ elementary, we can compute derivative code for $\bar{f}$ in the same way as we did in connection with Algorithm 1-2.

7. Within L replace all occurrences of $\bar{f}$ by $v_{\bar{f}}$. Also, within the derivative code generated in step 6, substitute an appropriate variable v_e for any expression e which has already been reduced. Next make appropriate additions to the set RC of region constants and to the induction sets IV. Moreover, within the derivative code generated in step 6, mark each node n 'reducible' if the subexpression text(n) is formed from a derivative code subpattern specially underlined in our D table. Recall that such underlining indicates that further reduction is necessary for FD to be profitable. After doing this, reduce all

such subexpressions using recursive application of Algorithm 2.

8. For each pattern variable x in f in which $\bar{x}$ is a generated variable holding the value of a reducible subexpression e of $\bar{f}$ available in L, if $\bar{x}$ is not used within derivative code generated for $\bar{f}$ then

i. Remove all derivative code (previously generated) which keeps the value of $\bar{x}$ current.

ii. Remove the initialization of $\bar{x}$ from the prologue for L.

iii. Make appropriate corrections to RC and N.

iv. Replace all occurrences of $\bar{x}$ in L by occurrences of e.

On the other hand, if the generated variable $\bar{x}$ is used within derivative code for $\bar{f}$, prompt the user so that he can supply a unique variable name to be used in place of $\bar{x}$.

To illustrate the application of semiautomatic FD to SETL, we again consider the expression (10) which we assume is executed repeatedly within a program loop L. A user of our proposed interactive system could select (10) or any of its reducible subexpressions (marked by Algorithm 1) for reduction. Suppose that in order to conserve space he chooses to reduce the full expression (10) by issuing the command,

(11) $FD, 15, Diff = {x ∈ (T - S)|f(x) = q} - Q

After validating (11), Algorithm 2 will arrange the reducible subexpressions of (10) in postorder as follows: (1) $c_1 = T - S$, (2) $c_2 = \{x \in c_1 | f(x)=q\}$, and (3) $c = c_2 - Q$. Then we first reduce c_1 which is matched by form 3 of the SETL F table (cf. Appendix C(ii)). To do this, we begin by inserting the assignment

(12) $$c_1 := T - S;$$

at the end of the prologue for L. Since T is a region constant and S is an induction variable which undergoes a single modification S := S - Δ at a program point p in L, the only update code for c_1 will be the prederivative

(13) $$c_1 := c_1 + \Delta * T$$

(generated by the macro found in the D table entry D(x2,3) of Appendix C(ii)) which will be inserted just before p. Next,all occurrences of T - S within L are replaced by occurrences of c_1. Then the identifier c_1 is added to the induction variable set IV_1.

At this point Algorithm 2 will select c_2 (which is matched by form 5 in the F table) for reduction. The assignment

(14) $$c_2 := \{x \in c_1 | f(x) = q\}$$

is inserted at the end of the prologue, just after (12), and the prederivative code

(15) $$c_2 := c_2 + \{x \in (\Delta * T) | f(x) = q\}$$

(corresponding to the change (13)) is placed just prior to (13). Then we replace occurrences of $\{x \in c_1 \mid f(x) = q\}$ by occurrences of c_2, and include c_2 within the induction variable set IV_1. However, since c_1 is not used in the derivative code (15) for c_2, all uses of c_1 within L are eliminated. That is, Algorithm 2 will delete the derivative code (13) within L, remove the initialization (12) from the prologue, and replace all occurrences of c_1 within L and its prologue by occurrences of T - S. The identifier c_1 will also be removed from IV_1.

Now the expression c originally selected for reduction by the directive (11) can be processed. Algorithm 2 will first place the initialization,

(16) $\qquad c := c_2 - Q;$

just after (14) in the prologue. It will then examine the D table entries associated with elementary form 3 of the F table and determine the derivative code for c relative to the change (15) in c_2 and the change $Q := Q + \Delta$ in Q. This will lead to insertion of the prederivative code

(17) $\qquad c := c + (\{x \in (\Delta * T) \mid f(x) = q\} - Q);$

just prior to (15). The update correction

(18) $\qquad c := c - (\Delta * c_2);$

will be placed immediately before the modification to Q. After substitution of occurrences of c for occurrences

of c_2 - Q the identifier c will be placed in both IV_1 and IV_2. Note that the fact that c_2 is used within (18) prevents Algorithm 2 from avoiding the reduction of c_2 (as was done previously in the case of c_1). Therefore, the system will request a user supplied name to replace c_2. If we suppose that this name is Temp, then all occurrences of c_2 within L and its prologue will be replaced by Temp. The reduction procedure completes its work by replacing all occurrences of c by the user supplied name Diff given in (11).

In consequence of these actions the end of the prologue to L will contain the following code.

(19)
```
Temp := {x ∈ (T-S) | f(x) = q};
Diff := Temp - Q;
```

Within L, the derivative code inserted just before the change to S will be

(20)
```
Diff := Diff + ({x ∈ (Δ * T)|f(x) = q} - Q);
Temp := Temp + {x ∈ (Δ * T) | f(x) = q};
```

while the update code inserted before the change to Q will be

(21)
```
Diff := Diff - (Δ * Temp);
```

The reader should note that instead of using (18), we might have used either of the following alternative

predederivative codes: $c := c - \{x \in \Delta \mid x \in T \ \& \ x \notin S \ \& \ f(x)=q\}$ or more simply, $c := c - \Delta$; either of these alternatives would eliminate the troublesome use of c_2 in (18) and would make it possible to avoid reduction of c_1 (thereby conserving space). A capability to choose between competing derivative code alternatives might be a useful future extension to Algorithm 2.

In the following chapter, we will extend the techniques just described to obtain an implementation of FD for general expressions. We will also study SETL FD more closely in order to prepare for several case studies of algorithms derived by FD and other transformations.

IV. IMPLEMENTATION DESIGN FOR FORMAL DIFFERENTIATION OF EXPRESSIONS CONTINUOUS IN SOME OF THEIR PARAMETERS

A. Introduction

In considering formal differentiation of expressions involving discontinuity variables, we face complexities which cannot be handled without extending the simple FD framework of Chapter III (C). The additional information needed to specify patterns for elementary discontinuous expressions, for modifications to variables of such expressions, and for associated update rules requires major adjustments to the simple structure of the F and D tables given in III (C). Moreover, a full FD system must cope with a greater number of relevant basic expressions, a more complicated assortment of induction variables, and a host of competing alternative transformations whose potential for program improvement is often unpredictable. All these factors make it awkward if not impossible to use FD algorithms utilizing easy variants of the F and D tables of the preceding chapter. However, in this final chapter we will modify those tables to support an FD implementation design fashioned around a relatively straightforward extension to the methods of III (C).

Although a fully automatic FD implementation is conceivable (cf. [W1] for a discussion of FD at the PASCAL

level), we will not consider this but instead will augment the semiautomatic FD techniques described in the last chapter. Our extended SETL FD system will incorporate many of the transformations described in Chapter II (D) within an extension of the interactive program impvovement facility described in Chapter III (A,B). The use of the proposed system will be illustrated by considering and improving several sample SUBSETL programs.

B. Semiautomatic Formal Differentiation of Discontinuous Expressions.

In this chapter we consider general expressions

(1) $f(x_1,\ldots,x_k,x_{k+1},\ldots,x_n)$

continuous in some of their variables $x_1,\ldots,x_k$, and discontinuous in the remaining variables. To differentiate expressions (1) in a program loop L, the general reduction method of Chapter I and the particular transformations of Chapter II, D must accomplish two related tasks:

1. Store separate values of (1) in a map $\bar{c}(x_{k+1},\ldots,x_n)$ defined on entrance to L.

2. Keep $\bar{c}$ available throughout L by updating $\bar{c}$ whenever variables on which $\bar{c}$ depends vary in L.

If each stored value $\bar{c}(x_{k+1},\ldots,x_n)$ depends continuously on changes to $x_1,\ldots,x_k$, and if each derivative of $\bar{c}$ involves an iteration over only a small portion of the domain

of $\bar{c}$, we say that $\bar{c}$ is continuous relative to the continuity variables of (1) and that $x_{k+1},\ldots,x_n$ are removable discontinuities. This will usually imply that the execution cost of the derivative code introduced by the second step noted above will be low relative to a single calculation of (1).

In the system to be described we will accordingly restrict reduction candidates to those expressions (1) all of whose discontinuity variables are removable. We will also limit our reduction methods to those explored in Chapter II (D) (with the exception of memo function techniques); these offer a high likelihood of attaining program improvement under circumstances recognizable by easy analysis. As in the set former case studied in Chapter II, we aim at order of magnitude speedups.

An implementation design for semiautomatic FD for discontinuous expressions can be based on the semiautomatic approach described in Chapter III. To find reducible expressions (with discontinuities allowed) in a loop L, we will make use of a marking algorithm, Algorithm 1', closely related to Algorithm 1 of Chapter III. Algorithm 1' proceeds by first determining the elementary reduction candidates $Cands_0$ (these are reducible expressions having no reducible subexpressions) before gathering up the remaining nonelementary ones. We will also discuss a reduction algorithm which we call Algorithm 2' (based largely

on Algorithm 2 of Chapter III) for differentiating expressions marked by Algorithm 1'.

Both these algorithms make use of a table F of elementary forms and a derivative code table D in a way similar to the related algorithms of Chapter III. However, to deal with the new problem areas raised by discontinuities, we will make use of more complicated pattern constructs, and more sophisticated matching and macro expansion operations. As we shall see, the major differences between the FD design of Chapter 3 (C) and our new design for discontinuous expressions will be localized in pattern handling. These differences involve format changes in our F and D tables and new versions of Match and Expand.

As in the case of completely continuous expressions, reduction candidate expressions involving discontinuities can be recognized as those which are formed by composition and parameter substitution from a finite collection F of elementary reducible forms. This recognition problem is handled primarily by our new Match routine (presented in Appendix E (V)) under the control of Algorithm 1'. Thus, Match will not only perform the simple syntactic kind of matching between a pattern f and an expression e (as was done previously), but it will also check a variety of restrictions imposed on each matched subexpression $\bar{x}$ of e, where x is a pattern variable of f. Consequently, when Match(e,f,Pfunc) holds, we know that e is reducible.

Although the aforementioned restrictions on subexpressions $\bar{x}$ of e are fairly uniform and easily expressed for completely continuous expressions, this is not the case for discontinuous expressions. Indeed, for discontinuous expressions, it will be convenient to test these restrictions by executing boolean valued function procedures Restrict(x,f) defined for each form $f \in F$ and for each pattern variable x in f. During a matching operation using a pattern f, and just after the pattern variable x of f is matched to a subexpressions $\bar{x}$, Restrict(x,f) must be executed and return *true* in order for matching of x to succeed.

We can categorize pattern variables x into three basic types -- discontinuity, continuity, and special parameters. If x is a discontinuity parameter than Restrict(x,f) will return *true* only if $\bar{x}$ consists entirely of free variables of $\bar{f}$ and also if $\bar{x}$ is not a region constant.

In the case where x is a continuity parameter, we can usually restrict $\bar{x}$ by the following predicate which Restrict(x,f) computes: $\bar{x}$ is a region constant expression *or* $\bar{x}$ is an induction variable belonging to IV(x,f) *or* $\bar{x}$ is an induction expression for f and x. Note here that the sets IV(x,f), essentially serving the same purpose as in the last chapter (cf., p. 392), can be computed for every f in F and each continuity parameter x in f. General routines to compute the set RC of region constants and the IV sets are given in Appendix E (iv).

When x is a special parameter, Restrict(x,f) must be programmed in a highly particular and unsystematic way. As an example, consider the special parameter K occurring as part of the following setformer pattern,

$$f = \{y \in xl \mid K\} .$$

For this case, we might want Restrict(K,f) to implement the following predicate: $\bar{y}$ is a variable occurring within $\bar{K}$ & for each variable g occurring in $\bar{K}$ one of the following conditions must hold:

1. g is a region constant.
2. $g = \bar{y}$.
3. g is a map variable which only occurs in K as a map retrieval involving $\bar{y}$ and is also an induction variable which can only vary by indexed assignments.

In actuality our patterns will only make use of a very few special procedures of the kinds just mentioned.

Since matching and expansion operations using a pattern P will always visit the nodes of P in postorder, we can specify when a particular procedure *pname* should be executed during either of these operations by inserting the term !*pname* at an appropriate place in P. During matching, procedures will usually be used for validating expressions matched to pattern variables. Thus, we will frequently insert a procedure name *procname* within a pattern immediately after the occurrence of a pattern variable *patname*, and will allow *procname* to refer to *patname*.

By allowing patterns to contain procedure names, we gain considerable power. However, in order to provide a practical pattern handling capability, it is necessary to include a few additional features. The routines Match and Expand shown in Appendix E (V) implement these features. In particular, these routines can handle the intricate patterns specified in Appendix C (iii) for the FD tables used with our SETL implementation design discussed in the next section.

For practical reasons, it is of considerable importance to allow a single pattern to match different variants of the same expression. To achieve this, we allow patterns to be built using alternation of subpatterns. We borrow SNOBOL notation for this. Using alternation, we can specify the entire elementary form table F used for FD as a single pattern,

(2) $$F = Form_1 \mid Form_2 \mid \ldots \mid Form_n$$

If Pfunc is the pattern variable map constructed during matching, then failure to match an alternand of F causes Pfunc to be restored to its previous value just before F is matched. Note that alternation provides a mechanism for choosing between competing transformations.

As a notational convenience, we allow pattern names to be associated with pattern expressions. This is achieved by assignments of the form

(3) <pattern name> = <pattern expression>

This feature allows us to use pattern names (in place of the pattern expressions they represent) as part of pattern expressions. It may be convenient to use several assignments of the form (3) for synthesizing a single pattern expression.

We will sometimes need to use a pattern which matches all of the components of a parameter list of arbitrary size. Such a pattern can be specified using the following recursively defined pattern name,

(4) Params = q3. ',' Params | q3.

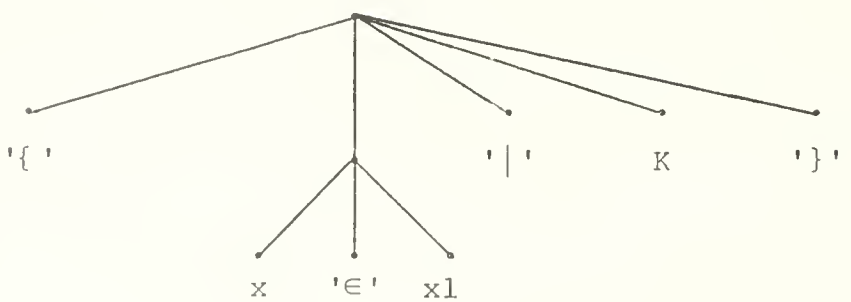
where the comma in quotes is a literal and q3. is a special pattern variable. The pattern assignment (4) borrows the notion of recursive pattern definition from Snobol. (Note that the right-hand side occurrence of Params in (4) is treated in the same way as a Snobol unevaluated expression). The period appearing immediately after the pattern variable q3 denotes that each time q3. is encountered during matching, a unique pattern variable (which we call an instance of q3.) is generated. Each such instance will have the form q3i where i-1 is the number of previous instances generated. When matching succeeds for q3i, $\overline{q3i}$ will refer to the text which is matched. If matching fails a previous system state s will be restored (i.e., the pattern variable map will be restored) and instances of q3. generated after the state s was last saved are lost.

Furthermore the underlying counter #q3 which we use to maintain the number of q3. instances is also restored. For example, if the left alternand of (4) fails in matching the comma just after q3i has been generated, the association between q3i and $\overline{q3i}$ will be destroyed, i will be reset to i-1, and matching will proceed with the right alternand of (4).

Some of the remaining features of our pattern facility are illustrated in the following pattern definition used for matching setformers,

(5) Form = ['{' [x '∈' x1] '|' [K] '}']

Note that the pattern expression (5) uses quoted symbols to denote literals, pattern variables x and x1, the pattern name K, and predecessor formation brackets used to express tree strcture for (5); this corresponds to the following tree representation:



In (5) we intend K to be a pattern name which matches a conjunction of terms in a rather general way. If we use a pattern name conj for matching conjuncts then K can be defined recursively as follows:

(6) K = [Conj] '&' K | [Conj]

where Conj is defined by the rule,

(7) Conj = [F8. '(' [x] ')' '=' '0'] |
K5. |
! x'∈' x2 |
! K4 '∈' [F2 '(' [Params] ')'] |
q4. '∈' [F3. '(' [x] ')']

The special symbol ! appearing in (7) triggers execution of a 'built-in' procedure whenever this symbol is encountered during matching. The effect of this procedure is to set up a gate for each occurrence of the symbol !. Initially, all such gates are 'open'. During matching, whenever a closed gate is encountered failure occurs; whenever an opened gate is reached, matching proceeds through the gate but leaves the gate closed. Since we store the state of each gate within the pattern variable map Pfunc, when failure occurs and a previous state of Pfunc is restored, closed gates may become reopened. In connection with the pattern (5), the preceding rules imply that any set former matched by the pattern (5) can have within its boolean subpart at most one conjunct matched by x '∈' x2 and one conjunct matched by K4 '∈' [F2 '(' [Params] ')'] .

To adapt (5) for matching differentiable expressions, we must modify (5) by inserting procedure names. Suppose that Dvar, Cvar, and Svar are boolean valued procedures

which validate discontinuity, continuity, and special parameters respectively. (Recall that we sketched the logic for these procedures earlier in this section.) Then within the pattern expressions (4), (5) and (7) we should insert !Dvar after each pattern variable whose name begins with the letter q; insert !Svar after each pattern variable beginning with the letter K; insert !Cvar after all remaining pattern variables with the exception of x.

After (5) is altered in this way, it will be able to match parse trees for an assortment of reducible SETL set formers. For example, (5) matches the parsed form of the following SETL setformer.

```
(8)     {x ∈ s|t ∈ g(x) & f(g(x + z)) = 0
                & x**2 ∈ h(t+b,a) & d*t ∈ f(x)
                & x ∈ Q & f(x) = 0}
```

The pattern variable map Pfunc which results from matching (5) to (8), will associate pattern variables of (5) with the text (8) in the following way:

(9) $\bar{x}$ = x, $\overline{x1}$ = s, $\overline{q41}$ = t, $\overline{q42}$ = d*t, $\overline{F31}$ = g, $\overline{F32}$ = f, $\overline{K51}$ = f(g(x + 2)) = 0, $\overline{x2}$ = Q, $\overline{K4}$ = x**2, $\overline{F2}$ = h, $\overline{q31}$ = t + b, $\overline{q32}$ = a, $\overline{F81}$ = f .

Once defined, Pfunc can be used to trigger macro expansion operations which produce code to reduce (8).

We treat macro expansion as the inverse of pattern matching, so that all of the operations just described for pattern matching can be used for macro expansion. After matching a pattern P to a parse tree T, we obtain a map Pfunc which associates pattern variables x of P to a matched subtree Pfunc(x) of T. Macro expansion will produce T starting from P and Pfunc. (Of course, it is not at all practical to force the converse to hold.)

A few special features of macro expansion must be noted. Suppose we use the map Pfunc which results from matching (5) to (8) and expand the pattern Exp defined below,

```
(10)        Exp = [E * '(' [Params] ')'] | E*
            Params = Param',' Params | Param
            Param = q3. | q4.
```

The symbol * occurring within (10) has special significance only in connection with patterns used for macro expansion. In the case of (10), when E* is encountered during expansion, a new program variable name *newname* and a new blank atom n are generated, where Leaf(n) = *True* and Label(n) = *newname*. We then associate the pattern variable E with n by making the SETL assignment Pfunc(E) := n.

As noted in Chapter 3, expansion proceeds by visiting the nodes of a pattern tree in postorder. Recall that expansion of a pattern tree P and a map Pfunc produces a new parse tree T which is formed from P essentially by

replacing each pattern variable x in P by a subtree Pfunc(x).

We also permit expansion of patterns such as (10) formed using alternation. To see how alternation works, we first note that a subpattern P' of a pattern P can only fail during expansion when P' contains a pattern variable not in the domain of the pattern variable map Pfunc. Failure causes Pfunc as well as the expanded parse tree T to be restored to their values just before expansion of P'. If P' is an alternand of a subpattern P' | Q, failure of P' will cause expansion to proceed with Q.

Dotted elementary variables such as q3. occurring in (10) behave the same way in expansion as in matching. When q3. is first encountered, expansion will treat q3. in the same way as q31, the first instance generated by q3. . After the i-1'st instance q3i-1 of q3. is expanded successfully and q3. is again encountered, we attempt to use the next instance q3i.

As an example, note that expansion of (8) using the Pfunc map defined by (9) yields the following expression,

(11) $\bar{c}(t+b,a,t,d*t)$

where $\bar{c}$ is a new variable name generated during expansion. As part of our FD procedure we will use the map retrieval operation (11) to replace the occurrence of (8) being reduced.

We can now give an example illustrating the structure of our derivative table D. Suppose that within this table, the entry D(xl,Form) for the elementary form (5) and pattern variable xl is a set which contains the triple [mod,preD,]. Let the modification pattern mod be defined by the rule

(12) mod = [xl ':=' [xl '+' Δ] ';']

and the prederivative pattern by the following pattern definitions,

(13) preD = ['(' '∀' [iterator] ')' [Add] 'end' '∀' ';']

iterator = iterpart | [K]

iterpart = [iter] ',' iterpart | [iter]

iter = !x '∈' '(' [Δ - xl] ')' |

!w2* '∈' ['{' [u* '∈' ['Project' '('

[#q3 ',' F2] ')']] '|'

[K4 '∈' [F2 '(' [u] ')']] '}'] |

w3.* '∈' [F3. '(' [x] ')']

K = [Conj] '&' K | [Conj]

Conj = !x '∈' x2 |

[F8. '(' [x] ')'] '=' '0' |

K5.

Add = P* = [E '(' [Params] ')'] ':='

[P '+' '{' [x] '}'] ';'

Params= Param ',' Params | Param

Param = !w2 | w3.

A few clarifying remarks are needed to explain this example. The subpatterns underlined within iter match parsed expressions which must be further reduced. The pattern variable #q3 occurring within iter is treated as a literal number whose value is the number of instances of q3. occurring within the map Pfunc used to expand preD. Finally the use P*= of the pattern variable P within add causes Pfunc(P) to be defined as the root of the subtree expanded from the pattern tree occurring just to the right of P*=.

We can use (12) and (13) to illustrate how we determine a derivative for (8) relative to a change s := s + p(y). Starting with the map Pfunc which becomes available after expansion of (10), we will match the pattern mod to the change in s. Pfunc will reflect the successful match by associating $\bar{\Delta}$ with p(y). Next we will use Pfunc to expand PreD. The prederivative code which results is

```
(14) (∀x ∈  (p(y)-s), u ∈ {v ∈ Project(2,h)|x**2 ∈ h(v)},
       w ∈ g(x), z ∈ f(x)|x ∈ Q & f(x) = 0 & f(g(x+2)) = 0)
       c̄(u, w, z) := c̄(u, w, z) + {x};
     end ∀;
```

where u, w, and z are new names generated for the pattern variables w2, w31, and w32 respectively.

A more formal description of our pattern notation can be found in Appendix C (iii). With each pattern specified

using our nonprocedural pattern language, we can associate a pattern tree. The rules of correspondence between pattern specifications and pattern trees are found in Appendix E (v). The patterns specified in Appendix C (iii) make comprehensive use of our pattern language. We handle these patterns by compiling them into pattern tree form and using the Match and Expand routines shown in Appendix E (v).

The major differences between FD implementations for entirely continuous expressions and for discontinuous expressions are found in our pattern language. A few additional differences will be pointed out in the following discussion concerning a general FD implementation.

A D table, much like that of the FD framework of Chapter III can play essentially the same role in the present formulation. For each elementary form f in F and every continuity parameter x of f, we again use a set D(x,f) of triples [mod, preD, postD] where mod is a parameter change pattern, PreD stands for a prederivative code macro, and postD a post derivative code macro. As before, we will still use an F table together with our D table to find induction variables, to find an initial set of reduction candidates, and to expand this initial set to a more general set of reduction candidates by means of the following variant of Algorithm 1.

Algorithm 1'.

1. Find the set RC of region constants and the sets of induction variables.

2. Arrange the nodes of L in postorder; i.e., perform the assignment T := Postorder(n); where n is the root node of L and Postorder is the SETL procedure given in Appendix E (iv).

3. For all $n \in T$ such that $\exists f \in F$ in which Match(n,f,Pfunc) holds perform step 4.

4. Place n in each of the induction sets for which the matched expression $\bar{f}$ is an induction expression. Use a map Reduce to record the values of f and Pfunc at n by executing the assignment Reduce(n) := [Pfunc,f]. (Reduce will be used in connection with the reduction routine Algorithm 2' to be described later.) Finally, mark n 'reducible'.

Once algorithm 1' has executed, all reducible expressions (with or without discontinuities) will be marked. Note that the postordering in step 3 ensures that we visit a reducible expression e only after first visiting all reducible sub-expressions of e. This is critical to our algorithm since determining reducibility for e depends on establishing that reducible subexpressions of e belong to appropriate induction sets -- analysis which should be done prior to examination of e.

Also, observe that the routine Match invoked in step 3 is the augmented matching procedure given in Appendix E (v).

It handles tree representations of patterns found in F, and performs various checks in addition to the purely syntactic tree matching operations used in Chapter 3 (where only purely continuous expressions are handled). In particular we allow Boolean function code procedures to be included as part of our pattern structures. When these procedures are encountered during matching, they are executed and must return *True* for matching to succeed.

It is worthwhile to make one more remark in connection with Algorithm 1'. By using the induction variable procedure shown in Appendix E (iv), Algorithm 1' could be implemented in a largely language independent manner. However, this procedure constructs induction variable sets IV(x,f) associated with every continuity parameter x of every elementary form f, and as is noted in the cases of Fortran and SETL (cf., Chapter 3 (2.3, 3.2)), this approach can be too costly in space. Thus, for each particular FD implementation we prefer to work out *ad hoc* induction variable routines (as we did for Fortran and SETL) which classify induction variables into fewer categories than the general procedure just noted. Moreover, for each separate implementation of Algorithm 1' we may have to redesign step 4 where we classify induction expressions according to the same categories defined for induction variables.

Algorithm 1' prepares for the semimanual selection and automatic reduction of expressions found in a program loop L. This process of selection and reduction is

accomplished by Algorithm 2' whose logic varies only slightly from that of Algorithm 2 of Chapter 3.

When a user selects an expression EXP for reduction by issuing a command

(15) $FD, LOOP#, NAME = EXP

Algorithm 2' will determine the discontinuities of EXP, also the arity of the map NAME which will store values of EXP, also the code initializing NAME which becomes part of the prologue of the loop designated by LOOP#, and finally will determine the derivative code to be inserted into the loop. Algorithm 2' will also determine which subexpressions of EXP to reduce in order to make differentiation profitable. Having said all this, we now outline Algorithm 2'.

Algorithm 2'.

1. For the command (15) to be valid, NAME must not be an existing program variable and LOOP# must refer to a program loop L which contains the expression EXP at a node n marked 'reducible' by Algorithm 1'.

2. If the validation check (1) is passed we order the reducible subexpressions of EXP in postorder by executing

Cands := [x ∈ Postorder(n) | Marked(x) = 'reducible']

3. For each node t selected from Cands, assign Reduce(t)(1) to f, compute Pfunc by executing Match(t,f,Pfunc), and perform steps 5-8.

4. Halt.

5. Generate a unique name $v_{\bar{f}}$ for the variable which will keep the value of $\bar{f}$ available in L. If $\bar{f}$ has discontinuities then $v_{\bar{f}}$ will be a map storing separate values of $\bar{f}$ each of which gives constant values to the discontinuities; otherwise, it will be a nonmap variable storing only the single value of $\bar{f}$. In either case we must initialize $v_{\bar{f}}$ at the end of the prologue for L.

6. For each induction variable $\bar{x}$ in $\bar{f}$, and for each program point $p \in Defs(\bar{x})$ at which the value of $\bar{x}$ undergoes change, insert derivative code which keeps $v_{\bar{f}}$ available in L. Derivative code can be generated in essentially the same way as in Algorithm 2 despite possible occurrences of discontinuities within $\bar{f}$, except that we must use the more powerful Match and Expand utilities given in Appendix E (v).

7. Within L replace each occurrence of $\bar{f}$ by the map retrieval $v_{\bar{f}}(\bar{q}_1,\ldots,\bar{q}_n)$ where $q_1,\ldots,q_n$ are the discontinuity parameters of f; however, if $\bar{f}$ has no discontinuities it can be replaced by a simple variable $v_{\bar{f}}$. Within the derivative code generated in step 6, replace any expression e which has already been reduced by an appropriate simple variable v_e when e is entirely continuous or by the map retrieval $v_e(b_1,\ldots,b_m)$ where $b_1,\ldots,b_m$ are the discontinuities of e. Next make appropriate additions to the set RC of region constants and to the induction sets IV. After this, mark each node n 'reducible' if n is introduced by the

derivative code in step 6 and requires further reduction. Reduce all such expressions using recursive application of Algorithm 2'.

8. This last step is identical to that of Algorithm 2.

Most of the details involved in actually implementing Algorithm 2' are either straightforward or follow immediately from discussion of Algorithm 2. However, two new problems arise. Initialization of the map variables $v_{\bar{f}}$ used in step 5 of Algorithm 2' for storing values of discontinuous expressions is more complicated and diverse than initialization in the case of completely continuous expressions. In step 7, we must use new techniques to identify different occurrences of the same discontinuous expression and to replace these occurrences by appropriate map retrieval operations.

To illustrate the initialization problem, we consider the following SETL example,

(16) $$\bar{c}(y_1,y_2) = [+: x \in f(y_1,y_2)]1$$

where y_1 and y_2 are the only discontinuity parameters of (16). When f is a programmer defined map then initialization for $\bar{c}$ involves a straightforward iteration over the domain of f; i.e.,

(17)
```
c̄ := nullset;
(∀[b1,b2] ∈ Project(2,f))
    c̄(b1,b2) := [x ∈ f(b1,b2)]1;
end ∀;
```

Suppose, on the other hand, that f is a compiler generated variable resulting from reduction of the setformer

(18) $$f(y_1,y_2) = \{x \in s \mid g(x) = y_1 \ \& \ h(x) = y_2\}$$

Reduction of (18) will cause the following initializing code,

(19)
```
f := nullset;
(∀w ∈ s)
     if [g(w), h(w)] ∈ Project(2,f) then
          f(g(w),h(w)) := f(g(w),h(w))+{w}; else
          f(g(w),h(w)) := {w};
     endif;
end ∀;
```

to be inserted at the end of the prologue for the loop L containing (18). To initialize $\bar{c}$, we could certainly use the code (17) inserted immediately after (19) in the prologue. However, it is profitable to exploit the incremental way in which f is defined in (19) in order to produce better initializing code than (17) for $\bar{c}$. Essentially, we can formally differentiate $\bar{c}$ relative to the changes to f occurring in (19). Inserting the prederivative code for $\bar{c}$ into (19), we come up with the following code which initializes both f and $\bar{c}$ together:

```
(20)          c̄ := nullset;
              f := nullset;
              (∀w ∈ s)
                   if [g(w),h(w)] ∈ Project(2,f) then
                        c̄(g(w),h(w)) := c̄(g(w),h(w))+1;
                        f(g(w),h(w)) := f(g(w),h(w))+{w}; else
                        c̄(g(w),h(w)) := 1;
                        f(g(w),h(w)) := {w};
                   endif;
              end ∀;
```

Moreover, if in step 8 of Algorithm 2' we make f dead on entrance to the optimization loop L, then it is easy to eliminate all assignments to f and uses of f within (20). After we replace the term Project(2,f) with Project(2,$\bar{c}$), all assignments to f within (20) can be removed as dead code. Note that it would not be so easy to eliminate f from the prologue in the case of (17).

The preceding illustration of incremental initialization exemplifies a general initialization technique based on the following idea. Whenever Algorithm 2' chooses to reduce an expression e, we know that e must be elementary and that all of its outermost reducible subexpressions are already reduced. These subexpressions are initialized in the loop prologue according to the postorder in which they are chosen for reduction. Whenever it is feasible, we will plan to initialize e differentially relative to the

incremental initialization to the last reducible subexpression e' of e occurring within the prologue.

To implement this method we will use an initialization code table Init. For each elementary form f ∈ F we associate a pair Init(f) = [def,Deriv] where def is a pattern to be used as a code macro for generating a straightforward initialization typified by (17); Derive is a partially defined map associating pattern variables x of f with sets Deriv(x) of pairs of the form [mod,PreD]. We will then use Deriv for differential initialization (cf. the discussion of (20)), in which the pairs [mod,PreD] serve the same purpose as the entries of the D table.

To determine initialization code for an expression e we take the following steps: Suppose that Match(e,f,Pfunc) holds. Then we execute the SETL assignment

[def, Deriv] := Init(f)

If e has no reducible subexpressions, or if its outermost reducible subexpression e' (whose initialization code occurs last in the prologue) is matched by a pattern variable x of f in which x ∉ *Dom* Deriv, proceed as in case 1 below; otherwise proceed as in case 2.

1. For this case, we use the nonincremental approach. This requires insertion of the code generated by Expand(def,Pfunc) at the end of the prologue.

2. To initialize e differentially, we first suppose that reduction of e' generates the map variable v_e, which

holds values of e'. Then within the initialization code for e', at each point p where $v_{e'}$ varies we must find a unique pair [mod,preD] belonging to the set Deriv(x) such that mod matches p using Pfunc, and we insert the prederivative code Expand(preD,Pfunc) just prior to p.

In Appendix C (iii), we show an Init table included as part of our SETL FD implementation to be discussed in the next section. Note that in this table, out of eleven basic forms contained in our F table (also shown in Appendix C (iii)), only Init(Form10) and Init(Form11) have nonnull second components. Nevertheless, the SETL case studies of Section D and Appendix F suggest that the differential initialization capability just mentioned has widespread utility.

It is of considerable importance for Algorithm 2' to be able to find all elementary reducible expressions in L whose values can be stored using the same variable. (Note that the variables which hold values of reduced expressions are generated by macro expansion during the initialization phase of FD.) We call such expressions 'similar', and can replace occurrences of similar expressions by occurrences of the same simple variable or of map retrieval terms using the same map variable.

Similar expressions which involve no discontinuities can differ from each other only in the names of bound variables. Similar expressions involving discontinuities

can also differ in their discontinuity subexpressions. For example, the SETL expressions

(21) {x ∈ s | f(x) = q * t}

and

(21') {y ∈ s | f(y) = p + b}

are similar when s and f are induction variables and q * t and p + b are discontinuity subexpressions. If we choose to reduce (21) in a loop L, then we will insert initialization code for (21) within the prologue to L and derivative code for (21) within L. If $\bar{C}$ is the map used to keep (21) available within L, then we can replace occurrences of (21) within L by occurrences of the retrieval $\bar{C}$(q * t). But since (21') is similar to (21) we can also replace occurrences of (21') by $\bar{C}$(p + b).

To formulate a general method which locates similar expressions and replaces them with simple variables or map retrievals, we note first of all that each expression e which Algorithm 2' selects for reduction is elementary, a fact which simplifies our task. Suppose that an expression e selected for reduction by Algorithm 2' is matched by the elementary form f and that the pattern variable map Pfunc is obtained by executing Match(e,f,Pfunc). We define the *decomposition* of e as the pair [f,Pfunc]. Let e' be some other elementary reducible expression having the

pair [f',Pfunc'] as its decomposition. Then e' and e are similar iff the following conditions hold:

(22) 1. f = f'

2. *Dom* Pfunc = *Dom* Pfunc'

3. Let Bnd(f) be the set of pattern variables of f which match to bound variables of an expression $\bar{f}$. For each nondiscontinuity parameter x belong to *Dom* Pfunc, let t be a tree formed from Pfunc(x) using the following substitutions: For each $y \in (\text{Bnd}(f) * (\textit{Dom}\ \text{Pfunc}'))$, replace all occurrences of subtrees Pfunc'(y) within Pfunc'(x) by the subtree Pfunc(y). After all of this is done, Equals(Pfunc(x),t) must hold, where Equals is the tree equality test given in Appendix E (ii).

Moreover, if we assume that two expressions g and h decompose into the pairs [f,Pfunc_g] and [f,Pfunc_h], we can test for similarity between g and h by executing Sim(f,Pfunc_g,Pfunc_h), where Sim is a boolean valued procedure which returns *True* if conditions 2 and 3 of (22) both hold.

Using the test (22) for similarity we can locate all expressions similar to an expression e which is chosen by Algorithm 2' for reduction. Then in step 7 of our algorithm, we can replace all occurrences of e and expressions similar to e by occurrences of appropriate simple variables or map retrievals. Let Similar(e) be the set containing e

and also containing all reducible expressions similar to e. Suppose that e is matched by basic form $f \in F$; suppose also that each expression $g \in Similar(e)$ can be decomposed into the pair $[f,Pfunc_g]$. To each map $Pfunc_g$ we then add the pair [E,t] (generated by expansion during initialization for e) in which E is the pattern variable whose name is Label(t) used to keep values of e available. Then, using a function Replace which associates each elementary form $h \in F$ with a replacement macro Replace(h), we can replace each occurrence of every expression $g \in Similar(e)$ with the term $Expand(Replace(f),Pfunc_g)$. (Cf., the Replace patterns in Appendix C (iii).)

Using the test for similarity that has just been described, we can also keep track of reduced expressions, and avoid redundant reduction of similar expressions. This is achieved by maintaining a set 'Reduced' of nonsimilar reduced expressions e each of which is represented by its decomposition $[f,Pfunc_e]$. The first time Algorithm 2' selects an expression e for reduction (in step 3) the set 'Reduced' is initialized to $\{[f,Pfunc_e]\}$ where f is the elementary form matching e, and $Pfunc_e$ is the pattern variable map obtained by matching f to e. Each subsequent time step 3 is performed, before choosing an expression e' (whose decomposition is [f',Pfunc']) for reduction, we check whether e' is similar to an expression already reduced. This can be done by performing the test

(23) $\exists Pfunc_g \in Reduced\{f'\} | Sim(f', Pfunc_{e'}, Pfunc_g)$

If the value implied by (23) is true, we must perform the indexed assignment $Pfunc_{e'}(E) := Pfunc_g(E)$ which retrieves from $Pfunc_g$ the name $\bar{E}$ of the variable which can keep e' available, and stores this name into $Pfunc_{e'}$. Then, we skip to step 7 of Algorithm 2' and replace all occurrences of e' by occurrences of $Expand(Replace(f'), Pfunc_{e'})$. For the case when (23) does not hold, we proceed to step 5 where we add the pair $[f', Pfunc_{e'}]$ to Reduced after first performing all the other subtasks specified for this step.

We conclude this section by describing some heuristics for constructing a table F of elementary forms which can be included as part of an FD implementation for a programming language P. These are as follows:

1. Define some minimal set E of applicative expressions in P, where each expression in E contains no more than one occurrence of the same variable.
2. For each expression $f \in E$ determine a set DS(f) of data structures each capable of storing values of f.
3. For each expression $f(x_1,...,x_n)$ belonging to E, for each data structure $d \in DS(f)$, and for each variable x_i, $i = 1,...,n$ determine empirically the kinds of modifications to x_i in which the value of f stored in data structure d can be updated at a cost which is much less than the cost of a fresh calculation of f. Let Cont be the set of pairs

[x_i,mod] where mod is a pattern representing a distinct kind of change in x_i on which f depends continuously.

4. For each pair p := [x,mod] belonging to Cont, let Removable(p) denote all the variables y of f (where $y \neq x$) which we know to be removable discontinuity variables relative to the change mod in the variable x.

5. Then, proceed in the manner indicated in the following code to compute a set 'Forms' containing pairs each of which can be used to construct an elementary expression form for our F table:

```
Forms := nullset;
(∀p ∈ Dom Removable)  /* Initialize Forms */
     Forms with [{p}, Removable(p)];
end ∀;
(while ∃ t ∈ Forms, q ∈ Forms|t ≠ q &
                        (t(1) + q(1)) ∉ Dom Forms)
     Forms := Forms - {[Conts,Disconts] ∈ Forms |
                        (t(1) + q(1) Incs Conts &
                        (t(2) * q(2)) = Disconts}
                      + [t(1) + q(1), t(2) * q(2)]};
endwhile;
```

6. Having computed Forms in the preceding step, we can construct an elementary pattern form $g(y_1,\ldots,y_n)$ for each pair [Conts, Disconts] belonging to Forms. We construct this pattern g using the same tree structure and literal symbols as occur in f. However, for $i = 1,\ldots,n$ y_i is treated as a pattern variable restricted according

to the following three cases:

(i) If x_i belongs to the set Disconts, then y_i is a discontinuity parameter.

(ii) If x_i is contained in *Dom* Conts, y_i is a continuity parameter, and the value of any variable $\bar{y}_i$ matched by y_i can only vary according to the modification patterns found in Conts$\{x_i\}$.

(iii) If neither (i) or (ii) applies, y_i must match a region constant.

(C) Implementation Design for SETL

In this section we build an implementation design of semiautomatic FD for SETL based on the general remarks of the preceding section. This design incorporates several of the transformations studied in Chapter II (D) and regards differentiation of general set formers

(1) $\{x \in s \mid K(x,t_1,\ldots,t_n)\}$

as being of primary importance.

As noted in Chapter II (D), the cost of executing (1) repeatedly in a loop L is proportional to $N \times (\#\ S) \times \text{Cost}(K)$, where N is the iteration count of L. The FD transformations applied by our system will keep the value of (1) available in L in either $(N + \#S) \times \text{Cost}(K)$ or $(N + (\#S) \times \log(\#S))\ \text{Cost}(K)$ elementary steps; and this will usually imply a speedup.

The FD design to be described in this section also aims to minimize the number of elementary reducible forms in F and the variety of transformations embedded in the D table without sacrificing power. This is achieved by using the single form (1) to handle set intersection, set differences, and other set theoretic operations. Moreover, we exploit the fact that whenever iterative expressions such as the arithmetic sum

(1') $[+: x \in s \mid K(x,q_1,\ldots,q_n)]e(x)$

are reducible, we can also reduce (1). But instead of using exhaustive elementary form and derivative entries for both (1) and (1'), we only need to specify all the entries related to (1) and the D table entries associated with the following two additional elementary expression forms,

$$[+: x \in s]e(x) \quad \text{and} \quad [+: x \in K(q_1,\ldots,q_n)]e(x)$$

for (1'). Thus, the F table given in Appendix C(iii) contains only 11 elementary forms.

To differentiate expressions in a Subset1 program P, we must preprocess P by transforming expressions of P into one of the 11 basic forms of F. Many of the 'preparatory' transformations used for this purpose are given in Appendix D (v). Since one FD transformation can potentially lead to another, the code produced by FD will contain expressions forced to match to the elementary forms of F, and will lack a readable quality. Thus, after FD is done, we will want to sweep up the leftover transformational debris by applying a battery of cleanup transformations (cf. Appendix D (vi) for examples).

For simplicity we will treat both preparatory and cleanup transformations semiautomatically. The sample user/system interaction for manipulating the Topological Sort program in Chapter III (B) exemplifies the effort required to apply transformations preparing for FD and

cleanup transformations afterwards.

To handle discontinuous SUBSETL expressions, our FD design will use the following mechanisms:

1. The procedure Regconst given in Appendix E (iv) for determining the set RC of region constants in L.

2. The routine 1SETL which is the same procedure as Algorithm 1' of the preceding section except for minor adjustments described later in this section. 1SETL determines the set Cands of reducible expressions in L.

3. The reduction procedure, 2SETL (essentially the same as Algorithm 2' given in Section B) for reducing members of Cands in L.

We will also make use of the following components tailored exclusively to Subsetl:

4. The F, D, Init, and Replace tables given in Appendix C(iii).

5. A procedure for determining sets of induction variables for Subsetl. (For the sake of efficiency, we avoid using the language independent induction set procedure found in Appendix E (iv).)

Since parts 1-4 of our FD design are discussed elsewhere, we begin with remarks about 5. Since a decision procedure for induction variables depends on knowing the types of variables, we first perform a type analysis. This can be based on either Tenenbaum's method [T1] or on some system of type declarations. Observation of the 11 elementary

expression forms in F and their associated derivatives in the D table indicates that only seven sets of induction variables are needed. These may be defined as follows:

IV_1 = {all set variables x that only undergo changes of the form, $x := x \pm \Delta$};

IV_2 = {all map variables f experiencing only indexed assignments in L};

IV_3 = {integer variables x which only change according to the rule, $x := x \pm \Delta$};

IV_4 = {all set valued maps $f(q_1,\ldots,q_t)$ which only have definitions of the form $f(q_1,\ldots,q_t) := f(q_1,\ldots,q_t) \pm \Delta$};

IV_5 = {positive integer valued 1-ary maps f(x) only undergoing modifications $f(x) := f(x) \pm \Delta$}, where Δ is a positive integer}.

IV_6 = {all set variables x that only undergo 'strict' set additions and deletions, $x := x \pm \Delta$};

IV_7 = {all set valued maps $f(q_1,\ldots,q_t)$ that only have definitions of the form $f(q_1,\ldots,q_t) := f(q_1,\ldots,q_t) \pm \Delta$ which are 'strict'}; .

Once these seven induction sets have been calculated, we can find all the reducible expressions in L by using lSETL. Although lSETL closely follows the logic of Algorithm 1', there are differences at step 4 in which

inductions expressions are determined. In this step lSETL will classify the nodes n which correspond to reducible expressions into the sets described above, according to the following cases which arise for Subset1:

1. n is matched by Form1 of the F table. In this case, if there are no discontinuities place n in IV_6 and IV_1; otherwise place n in IV_7 and IV_4.
2. n is matched by Form1, Form2, ..., or Form7. Place n in IV_7 and also IV_4.
3. n is matched by Form 8. n goes into IV_6 and IV_1.
4. n is matched by Form 9. Insert n into IV_3.
5. n is matched by Form 10. Put n into IV_5.
6. n is matched by Form 11. Add n to both IV_4 and IV_7.

lSETL must also take an extra precautionary measure in order to recognize expressions e which depend on a set or tuple valued variable x having multiple occurrences in e. As an example of such an expression, consider

(3) $$c = \{x \in s \mid f(x) \notin s\}$$

occurring in a program loop L in which f is invariant and s only varies by set additions of the form $s := s + \Delta$. In accordance the method of Chapter 3 (C), we first number the two occurrences s_1 and s_2 of s within (3). Next we examine the prederivative code $\overline{preD_1}$ and $\overline{preD_2}$

for (3) relative to the changes $s_1 := s_1 + \Delta$ and $s_2 := s_2 + \Delta$ respectively. The D table of Appendix C (iii) shows that there are two cases to consider — that in which the set addition s := s + Δ is strict (i.e., the predicate $s * \Delta = nullset$ holds just prior to the change in s) and that in which it is not strict. In the first case, $\overline{preD_1}$ and $\overline{preD_2}$ are

(4)
```
/* preD1 bar */
(∀x ∈ Δ | f(x) ∉ s2)
     c := c + {x};
end ∀
```

and

(4')
```
/* preD2 bar */
(∀y ∈ Δ, x ∈ {u ∈ x1 | f(x) = y})
     c := c - {x};
end ∀;
```

In the latter case, $\overline{preD_1}$ and $\overline{preD_2}$ are given by

(5)
```
/* preD1 bar */
(∀x ∈ (Δ - s1) | f(x) ∉ s2)
     c := c + {x};
end ∀;
```

and

(5')
```
/* preD2 bar */
(∀y ∈ (Δ- s2), x ∈ {u ∈ s1|f(x) - y})
     c := c - {x};
end ∀;
```

In both of these two cases we seek to arrange $\overline{preD_1}$ and $\overline{preD_2}$ together with s := s + Δ in an appropriate order which makes it convenient to replace all occurrences of s_1 and s_2 within $\overline{preD_1}$ and $\overline{preD_2}$ by occurrences of s. (Recall that this is an optimal form of (6), Chapter 3(C).) In the first case, this can be achieved by surrounding the code s := s + Δ by (4) and (4') in any order. However, since (5) and (5') both depend on s_1 and s_2, we cannot find an optimal derivative code placement which avoids an extraneous and potentially costly copy operation. Thus, when the latter case arises we will not want to mark (3) 'reducible'.

Setformers such as (3) are used widely enough in SETL algorithms to warrant a few additional remarks. It is possible to reduce (3) by taking either of the following two approaches.

1. Choose less efficient or less desirable derivative code which makes use of fewer parameters than the standard derivative code.
2. Recognize that uses of parameters within derivative code may be eliminated by transformation.

The first approach allows us to reduce (3) by noting that the derivative code (4) and (4') can be used to replace (5) and (5'). However, if we do this, c will no longer belong to IV_6, and an outer expression such as [+: x ∈ c]1 which depends on c may require more complicated reduction.

The second approach avoids the disadvantages of the first, but is trickier to apply. In the case of expression (3), we can observe that the occurrence of s_1 in (5') is contained within the setformer $c'(y) = \{u \in s_1 \mid f(x) = y\}$ which must be reduced. Consequently, after reduction of c', the unwanted occurrence of s_1 will disappear. Thus, we can treat $\overline{preD_1}$ as involving s_1 and s_2 and $\overline{preD_2}$ as involving just s_2. A profitable derivative for (3) can then be formed by using $\overline{preD_1}$ followed by $\overline{preD_2}$ which precedes the change to s. Moreover, we must place the derivative code for c'(y) in between $\overline{preD_1}$ and $\overline{preD_2}$.

We will consider the two approaches just described as future possibilities for an expanded FD system. In any case, we will abide by the following general rule: If an expression e is matched by a basic form f more than one of whose continuity parameters match different occurrences of x in e, then we will consider e undifferentiable if our reduction techniques (cf., the discussion following (3) of Chapter 3(C)) would fail to remove costly copy operations,

(2) $$x_{OLD} := x$$

from the derivative code for e. Although we have not provided a general decision procedure for doing this which works better than the obvious exponential method, expressions involving multiple occurrences of a single variable

are apt to be uncommon.

Once all reduction candidate expressions are found in L, a user can select a particular candidate, EXP, for reduction by issuing the following command.

(6) $FD,LOOP#,NAME = EXP .

This command will be passed to the formal differentiation transformation generators, and will be processed by 2SETL (which is essentially a renaming of Algorithm 2' of the preceding section) using the Subsetl tables F, D, Init, and Replace of Appendix C (iii), the sets RC, IV_1,IV_2,...,IV_7 and the map Reduce generated by lSETL.

In the next section we will illustrate our SETL FD design by presenting four program derivations as case studies. However, before doing this it may be helpful to make a few explanatory comments about the SETL F and D tables given in Appendix C (iii).

Because the setformer is a fundamental building block used to construct base forms of algorithms, the most important elementary form appearing in the F table is the setformer pattern Forml. Forml matches generalized set formers which cover many of the set former constructs studied in Chapter 2 (D). The boolean subparts of set formers matched by Forml must consist of a conjunction of terms T, each of which is matched by a conjunct pattern used in the definition of Forml. Recall from Chapter 2(D)

that differentiation of set formers is sometimes handled differently for the two cases when a particular kind of conjunct occurs only once or when it occurs several times within T (cf. (49) of Chapter 2 (D)). However, we have observed that whenever such a distinction is made for a conjunct pattern p, we can handle multiple occurrences of terms which match to p by preparatory transformations which reduce these multiple occurrences into a single occurrence. We enforce this procedure by designing Forml to allow p to match successfully no more than once. This is done by placing the special pattern operator ! just prior to p within the definition of Forml.

To illustrate the technique just mentioned consider the following set former,

(7) $\{x \in s \mid f_1(x) \in Q_1 \ \& \ f_2(x) \in Q_2 \ \& \ \ldots \ \& \ f_n(x) \in Q_n\}$

When n = 1, lSETL can recognize (7) as a differentiable expression matched by Forml. However, for n > 1, (7) will not be marked 'reducible'. Thus, to reduce (7) we must first manually select transformations which rename $f_i(x)$ and Q_i as 'shadow variables' f'(x,i) and Q'(i), i = 1,...,n. Next we transform the conjunction f'(x,1) ∈ Q(1) & ... & f'(x,n) ∈ Q'(n) into the following intermediate form.

(8) $[+: 1 \leq i \leq n \mid f'(x,i) \notin Q(i)]1 = 0$

which passes into the differentiable form

(8') $[+: y \in \{1 \leq i \leq n \mid f'(x,i) \notin Q(i)\}]1 = 0$.

(Cf. (45), (45'), (47), (47') and (49) of Chapter 2(D) for further discussion.)

Although the FD tables of Appendix C (iii) allow us to handle a wide variety of expressions and algorithms, these tables are by no means complete; indeed, our F table omits several elementary forms which generalize expressions studied in Chapter 2 (D) (cf., (49) of Chapter 2 (D)), and our D table lacks entries which could handle Rule 2. These omissions simplify our initial implementation design, and can be easily remedied in the future.

(D) Applications of Formal Differentiation to Algorithm Development

To come to terms with the question of how automatically FD can be applied, we consider a simple example — Knuth's Topological Sort. (This example is also studied by Earley, [E1].) The input assumed by this algorithm is a set s and a set of pairs sp representing an irreflexive transitive relation defined on s; as output, it produces a tuple t in which the elements of s are arranged in a total order consistent with the partial order sp. A concise SETL form of the algorithm is as follows:

```
(1)  t := nulltuple;
     (while ∃a ∈ s | (sp{a} * s) = nullset)
        t := t + [a];     /* tuple concatenation */
        s := s - {a};
     end while;
```

In Chapter 3 (B) we showed how a user of our proposed system could transform (1) semimanually into the following form which is better suited to FD:

```
(2)  1   t := nulltuple
     2   (while ∃a ∈ {x ∈ s|[+: y ∈ {z ∈ sp{x}|z∈s}]1 = 0})
     3      t := t + [a];
     4      s := s - {a};
     5   end while;
```

At this point a user could differentiate (2) by issuing the command,

(3) $FD, 2, Zrcount = {x∈s|[+: y ∈{z ∈ sp{x}|z ∈ s}]1 = 0}.

The system will have computed the sets RC = {sp} and IV_1 = {s}. Algorithm 1SETL will have marked the following expressions reducible:

$$c_1(x) = \{z \in sp\{x\} | z \in s\},$$

and

$$c_2(x) = [+: y \in c_1(x)]1 \quad ,$$

$$c_3 = \{x \in s \mid c_2(x) = 0\} \ .$$

Expression $c_1(x)$ matches basic form 11 of the F table (cf. Appendix C(iii)), $c_2(x)$ corresponds to form 10, and c_3 is of the first basic form.

Algorithm 2SETL, invoked by the user directive (3), will first validate this directive. Then, Zrcount will be reduced, starting from its inner and proceeding to its outer reducible subexpressions. To reduce the innermost subexpression $c_1(x)$, the system needs to differentiate c_1 with respect to the change s := s - {a} occurring at line 4 of (2), and also needs to initialize c_1 just prior to line 2. The system will use entry 11b in the D table to generate the following prederivative code,

```
(4)  (∀y ∈ ({a} * s), u ∈ {z ∈ Dom sp|y ∈ sp{z}})
          c1(u) := c1(u) - {y};
     end ∀;
```

The following initializing code will also be obtained from the Init table entry 11a:

```
(5)  (∀x ∈ Dom sp) c1(x) := {z ∈ sp{x} | z ∈ s};
     end ∀;
```

However, since the derivative table entry 11b stipulates that the subexpression $c_4(y) = \{z \in Dom\ sp | y \in sp\{z\}\}$ of (4) must also be reduced, the system will insert the following initializing code for c_4 at the end of the prologue to the *while* loop L,

```
(6)  c4 := nullset;
     (∀z ∈ Dom sp, y ∈ sp{z} | z ∈ Dom sp)
          if y ∈ Dom c4 then
               c4(y) := c4(y) + {z}; else
               c4(y) := {z};
          endif;
     end ∀;
```

(Note that (6) is based on entry 1 of the Init table.) No derivative code for c_4 is required, however, because c_4 is invariant in L. The system will request the user to supply a variable name for c_4. After all this the code sequence (4) will be transformed into a more efficient form,

```
(7)  (∀y ∈ ({a} * s), u ∈ c4(y))
          c1(u) := c1(u) - {y};   /* strict deletion */
     end ∀;
```

Next, proceeding from inner to outer subexpression of Zrcount, the reduction procedure aims to differentiate $c_2(x)$. Since c_2 only depends on the change in $c_1(u)$ which occurs within (7), entry 10b of the D table is applicable and yields the special prederivative,

(8) $\quad c_2(u) := c_2(u) - [+: x \in \{y\}]1$

This exploits the fact that the element deletion within (7) is *strict*, where we say that set addition $x := x + \Delta$ (or set deletion $x := x - \Delta$) is *strict* if the precondition $\Delta \cap x = \emptyset$ (respectively, $x \supseteq \Delta$) holds.

The system will now detect that c_1 has no uses within (8) and can therefore be eliminated. The initializing code for c_2 (obtained from entry 10a of Init), which is

(9) $(\forall x \in Dom\ sp)\ c_2(x) := [+: y \in \{z \in sp\{x\} | z \in s\}]1;$
end ∀;

replaces (5), and the assignment to $c_1(u)$ within (7) is removed.

Finally, c_3 is prepared for reduction. Its prederivatives, which are inserted within L, are

(10) $(\forall x \in (\{a\} * s) \mid c_2(x) = 0)$ /* change to s */
$c_3 := c_3 - \{x\};$
end ∀;

(cf., 1b of D) and

```
(10')      if u ∈ s & c2(u) = [+: x ∈ {y}]1 then
               c3 := c3 + {u};    /* change to c2 */
           endif
```

(cf., lhh of D).

Since both (10) and (10') contain uses of c_2 , c_2 will not be eliminated. Consequently, the system will request the user to supply a variable name for c_2. It will also initialize c_3 by inserting the following assignment (based on entry 1 of Init),

```
(11)       c3 := {x ∈ s | c2(x) = 0};
```

at the end of the prologue to L.

If we assume that the user supplies the name succ and count for c_4 and c_2 respectively, then the following much improved version of the topological sort (2) will be produced by one user directive (3):

```
(12) 1     t := nulltuple;
           /* prologue */
     2     succ := nullset;    /* successor map */
     3     (∀z ∈ Dom sp, y ∈ sp{z} | z ∈ Dom sp)
     4          if y ∈ Dom succ then
     5               succ(y) := succ(y) + {z}; else
     6               succ(y) := {z};
     7          endif;
     8     end ∀;
     9     (∀x ∈ Dom sp) count(x) := [+: y∈{z∈sp{x}|z∈s}]1;
    10     end ∀
    11     Zrcount := {x ∈ s | count(x) = 0};
           /* main loop */
```

```
12      (while ∃a ∈ zrcount)
13           t := t + [a];
14           (∀y ∈ ({a} * s), u ∈ succ(y))
15                if u ∈ s & count(u) = [+: x ∈ {y}]1 then
16                     zrcount := zrcount + {u};
17                endif
18                count(u) := count(u)-[+: x ∈ {y}]1;
19           end ∀;
20           (∀x ∈ ({a} * s) | count(x) = 0)
21                zrcount := zrcount - {x};
22           end ∀;
23           s := s - {a};
24      end while;
```

The topological sort shown in (12) can now be cleaned up in a way similar to the program manipulation performed on the second topological sort version of Chapter 3 (B). The code which then results is:

```
(13)      t := nulltuple;
          /* prologue */
          succ := nullset
          (∀z ∈ Dom sp, y ∈ sp{z})
               if y ∈ Dom succ then
                    succ(y) := succ(y) + {z}; else
                    succ(y) = {z};
               endif;
          end ∀;
          (∀x ∈ Dom sp) count(x) := [+: y∈sp{x}|y∈s]1;
          end ∀;
          zrcount := {x ∈ s | count(x) = 0};
          /* main loop */
          (while ∃a ∈ zrcount)
               t := t + [a];
```

```
        (∀u ∈ succ(a))
            if count(u) = 1 then
                zrcount := zrcount + {u};
            endif;
            count(u) := count(u) - 1;
        end ∀;
        zrcount := zrcount - {a};
    end while;
```

This final version of the topological sort algorithm will run in a number of cycles proportional to the number *nsp* of elements in the map *sp*. The original form (1) of the algorithm will require something like *nsp* * (#*s*)*(#*s*) cycles, which can be much larger. However, the symbolic chain of transformations going from (1) to (2) and from (12) to (13) is somewhat tedious (cf., Chapter 3). Moreover, it does not seem likely that these preparatory and cleanup transformations can be applied in a completely automatic way.

Some of the manual effort these transformations require might, however, be alleviated by integrating some of the transformations found in Appendix D (v) and (vi) as part of the FD algorithms. For example, since an existential quantifier Q contains the same variables which would appear in the set former F implied by Q, the marking algorithm lSETL can certainly determine whether F is differentiable or not. If it is, then lSETL can initiate transformation P6 of Appendix D(v) which will transform Q into

a form which exposes F. Likewise, after each of its steps, the reduction routine 2SETL can attempt to apply all of the cleanup transformations of Appendix D (vi) within the localized program regions just changed.

Another example closely related to the topological sort is the transitive closure algorithm found in [Schl]. This algorithm takes a set s and a multivalued mapping f as input. As output, it produces the smallest set s' which includes s and is equal to the image of f restricted to s'. A succinct SETL version of this algorithm is as follows:

```
(14) 1      (while ∃a ∈ s | s ≠ s + f{a})
     2           s := s + f{a};
     3      end while
```

In order to prepare (14) for FD a user might instruct the system to apply the following sequence of transformations:

1. Turn the predicate s ≠ s + f{a} appearing in line 1 into a more convenient form f{a} - s ≠ *nullset* by applying E4, D4, N4, and N1 of Appendix D (i).
2. P18, P8, and P6 of Appendix D (v) can then be used to transform the *while* loop predicate in a form suitable for FD.

The result of these steps is the following version of (14),

```
(15) 1      (while ∃a ∈ {x ∈ s|[+: z∈{y∈f{x}|y∉s}]1 ≠ 0})
     2           s := s + f{a};
     3      end while;
```

In (15), we note just as in the previous example that RC = {f} and IV_1 = {s}. Procedure 1SETL will mark the following expressions 'reducible':

$$c_1(x) = \{z \in f\{x\} \mid z \notin s\},$$
$$c_2(x) = [+: y \in c_1(x)]1 \quad , \text{ and}$$
$$c_3 \quad = \{x \in s \mid c_2(x) \neq 0\},$$

where $c_1(x)$ and c_3 differ (but only slightly) from their counterparts in the topological sort.

Then if a user issues the directive,

(16) $FD,1,Differ = {x ∈ s|[+:z∈{y ∈ f{x}|y ∉ s}]1 ≠ 0}) ,

2SETL will go through essentially the same steps as for (3). Of course, a slightly different derivative code sequence will be obtained for c_1. However, the reduction actions triggered by (16) will not lead to the optimally efficient code. This is because the prederivative of c_1 with respect to s := s + f{x}, i.e.,

(17) (∀y ∈ (f{a} - s), u ∈ {z ∈ *Dom* f |y ∈ f{z})
$c_1(u) := c_1(u) - \{y\};$
end ∀;

contains a hidden occurrence, $c_1(a) = f\{a\} - s$, of c_1. Unfortunately this occurrence will go undecected by 2SETL and c_1 will be prematurely eliminated as dead following the reduction of c_2. The code which results

will be correct, but it will fall short of the desired efficiency. Thus, it is better to handle (15) by taking smaller and more careful formal differentiation steps.

Another problem with (15) is that we cannot profitably differentiate the expression s + f{a} appearing in line 2 of (15). This makes the assignment at line 2 potentially inefficient.

To make (15) more suitable for FD, a user can apply the transformation P19 of Appendix D (v) to line 2. The code which results,

```
(18) (∀x ∈ (f{a} - s))
                s := s + {x}    /* strict addition */
     end ∀;
```

can then be transformed still further by turning f{a} - s into the canonical form {y ∈ f{a}|y ∉ s} which is reducible. (Transformation P18 of Appendix D (v) accomplishes this second task.)

Next, the user can issue the following FD directive,

```
(19)        $FD,2,Prout = {y ∈ f{x} | y ∉ s} .
```

The code which results is shown as follows:

```
(20)          /* Prologue */
              /* succ is supplied as the auxiliary map name
                 needed for reduction of cl */
     1        succ := nullset;
     2        (∀z ∈ Dom f, y ∈ f{z} | z ∈ Dom f)
     3             if y ∈ Dom succ then
     4                  succ(y) := succ(y) + {z}; else
     5                  succ(y) := {z};
     6             endif ;
     7        end ∀;
                   /* Prout corresponds to cl */
     8        (∀x ∈ Dom f)
     9             Prout(x) := {z ∈ f{x} | z ∉ s};
    10        end ∀;
              /* main loop */
    11        (while ∃a ∈ {x ∈ s|[+: z ∈ Prout(x)]1 ≠ 0})
    12          (∀x∈Prout(a))
    13             (∀z ∈ {x}, u ∈ succ(z))
    14                  Prout(u) := Prout{u} - {z};
    15             end ∀; /* prederivative of prout */
    16             s := s + {x};
    17          end ∀;
    18        end while;
```

At this point one additional user command,

```
(21) $FD,11,Differ = {x ∈ s|[+: z ∈ Prout(x)]1 ≠ 0}
```

will carry (20) into a penultimate version of (14). An efficient final form is subsequently reached by applying standard cleanup transformations.

It is useful to follow the FD actions triggered by (21). Analysis of the main *while* loop (line 11) of (20) will show that set s belongs in IV_1 and the map Prout is a member of IV_4. 1SETL will discover two reducible expressions, $c_2(x) = [+: z \in Prout(x)]1$ matching form 10 of F, and $c_3 = \{x \in s | c_2(x) \neq 0\}$ which is of the first elementary form with c_2 corresponding to the pattern F9. 2SETL will first attempt to differentiate the inner expression c_2 before it reduces c_3. Based on entry 10b of the D table, the system obtains the following prederivative code for c_2 with respect to the change to Prout at line 14 of (20),

```
(22)        c2(u) := c2(u) - [+: w ∈ {z}]1;
```

Note that, owing to the strict element deletion occurring at line 14, (22) will be generated by a special derivative code entry. The initialization code for c_2, obtained from entry 10a of Init, is

```
(23)        (∀y ∈ Dom Prout)
                c2(y) := [+: x ∈ Prout(y)]1;
            end ∀;
```

which is placed at the end of the prlogue. Finally, c_3 will be reduced as prescribed by entry 1a and 1jj of the D table.

The prederivative code for c_3 with respect to the change (22) is

```
(24)        if u ∈ s & c2(u) = [+: w ∈ {z}]1 then
                c3 := c3 - {u};
            endif;
```

The prederivative code to update c_3 relative to the strict element addition to s is

```
(25)        (∀y ∈ {x} | c2(y) ≠ 0)
                c3 := c3 + {y};
            end ∀;
```

Since (24) and (25) contains uses of c_2 , c_2 will remain as a reduced subexpression of c_3. The system will request the user to supply a name for c_2 and will also initialize c_3 at the end of the prologue. In response to a user request, $UNPARSE, at this point the system would print out the following more efficient version of (20),

```
(26)        /* prologue */
     1      succ := nullset;
     2      (∀z ∈ Dom f, y ∈ f{z}|z ∈ Dom f)
     3          if y ∈ Dom succ then
     4              succ(y) := succ(y) + {z};  else
     5              succ(y) := {z};
     6          endif;
     7      end ∀;
            /* Prout corresponds to c1 */
```

```
8     (∀x ∈ Dom f)
9          Prout(x) := {z ∈ f{x}|z ∉ s};
10    end ∀;
      /* count is the user name for c2 */
11    (∀x ∈ Dom Prout)
12         count(x) := [+: z ∈ Prout(x)]1;
13    end ∀;
14    Differ := {x ∈ s|count(x) ≠ 0};
      /* main loop */
15    (while ∃a ∈ Differ)
16         (∀x ∈ Prout(a))
17              (∀z ∈ {x}, u ∈ succ(z))
18                   if u ∈ s & count(u) = [+: w ∈ {z}]1then
19                        Differ := Differ - {u};
20                   endif;
21                   count(u) := count(u)-[+: w ∈ {z}]1;
22                   Prout(u) := Prout(u) - {z};
23              end ∀;
24              (∀y ∈ {x} | count(y) ≠ 0)
25                   Differ := Differ + {y};
26              end ∀;
27              s := s + {x};
28         end ∀;
29    end while;
```

To clean up the above code, one can apply the following transformations all of which are described in Appendix D: c_7 of Section vi will simplify [+: w ∈ {z}]1 occurring at lines 18 and 21 to 1; c_2 of the same section will turn the loop at line 24 into a conditional statement,

```
if count(x) ≠ 0 then
     Differ := Differ + {x};
endif;
```

A similar transformation applies to loop 17. Finally, the redundant element test z ∈ *Dom* f at line 2 can be eliminated by a variant of transformation s_2 of Section iii.

As a third example, we consider an algorithm which finds an interval in a flow graph. Input to this algorithm consists of a set V of nodes, an edge collection E represented as a mapping, and a root node H. The algorithm will output a set Int of nodes forming the interval in V whose header node is H. A base form SETL version of this algorithm can be written in the following way:

(27)
```
Int := {H};
(while ∃a ∈ (E[Int] - Int)|({x∈(V-Int)|a∈E(x)}
                    = nullset))
     Int := Int + {a};
end while
```

But in order to put (27) into a form suitable for FD, we change it by straightforward interactive program manipulation into the following equivalent form:

```
(28) 1     Int = {H};
     2     (while ∃ a ∈ {x ∈ [+: w ∈ Int]E(w)|x ∉ Int
                & [+: z ∈ {y ∈ V|y ∉ Int & x ∈ E(y)}]1 = 0})
     3        Int := Int + {a};
     4     end while;
```

Though seemingly more complicated than the previous two case studies, (28) can actually be differentiated using only one user directive,

```
(29)       $FD, 2, New = {x∈[+: w ∈ Int]E(w)|x ∉ Int
              & [+: z ∈ {y ∈ V|y ∉ Int & x ∈ E(y)}]1 = 0}.
```

To process (29), the system will note that IV_1 = {Int} and RC = {E,V}. lSETL will discover the following four reducible expressions:

$$c_1(x) = \{y \in V \mid y \notin Int \ \& \ x \in E(y)\}$$

$$c_2(x) = [+: z \in c_1(x)]1$$

$$c_3 = [+: w \in Int]\ E(w)\ ,$$

and

$$c_4 = \{x \in c_3 | x \notin Int \ \& \ c_2(x) = 0\},$$

each of them matching a different elementary form found in F.

The reduction procedure, 2SETL, will reduce the innermost expressions c_1 and c_3 first. The prederivatives of c_1 and c_3 relative to the modification of Int at line 3 of (28) are

```
(30)        (∀y ∈ ({a} - Int), x ∈ E(y) | y ∈ V)
                 c1(x) := c1(x) - {y};
            end ∀;
```

(obtained from entry 1e of D), and

```
(31)        c3 := c3+ ([+: w ∈ ({a}- Int)]E(w) - c3);
```

(30) and (31) will be inserted (in an arbitrary order) just prior to line 3. Within the prologue to L, 2SETL will insert the following code.

```
(31')       /* initialize  c1 */
            c1 := nullset;
            (∀y ∈ V, x ∈ E(y)|y ∉ Int)
               if x ∈ Dom c1 then
                  c1(x) := c1(x) + {y}; else
                  c1(x) := {y};
               endif;
            end ∀;
            /* initialize c3 */
            c3 := [+: w ∈ Int]E(w);
```

which makes c_1 and c_3 available on entrance to the while loop L.

Next 2SETL will reduce c_2. The prederivative of c_2 with respect to the assignment $c_1(x) := c_1(x) - \{y\}$ within (30) is given by the following code,

(32) $c_2(x) := c_2(x) - [+: w \in \{y\}]1;$

Since c_2 does not depend on c_1 , c_1 can be eliminated from L and from the prologue P to L. To do this, 2SETL first inserts initializing code for c_2. Because c_2 depends on c_1 , and because c_1 is defined incrementally within P, c_2 will be initialized by means of incremental entries 10b and 10c of Init. Once this has been accomplished c_1 can be removed from P and L.

2SETL is now ready to reduce c_4. However, before we describe the reduction steps used for this purpose, it is useful to pause and see what has already taken place. The state of the interval program after the transformations already noted is as follows:

```
(33) 1      Int := {H};
             /* prologue */
     2      c2 := nullset;     /* define c2 */
     3      (∀y ∈ V, x ∈ E(y)|y ∉ Int)
     4           if x ∈ Dom c2 then
     5                c2(x) := c2(x) + [+: w ∈ {y}]1; else
     6                c2(x) := [+: w ∈ {y}]1;
     7           endif;
     8      end ∀;
     9      c3 := [+: w ∈ Int]E(w);     /* define c3 */
             /* main loop */
     10     (while ∃a ∈ {x ∈ c3 | x ∉ Int & c2(x) = 0})
     11          (∀y ∈ ({a} - Int), x ∈ E(y) | y ∈ V)
     12               c2(x) := c2(x) - [+: w ∈ {y}]1;
     13          end ∀;
     14          c3 := c3 + ([+: w ∈ ({a} - Int]E(w) - c3);
     15          Int := Int + {a};
     16     end while;
```

But to continue: c_4 matches elementary form 1 in F, and it depends on parameters c_3, Int, and c_2 corresponding to pattern variables x_1, x_3, and F8 of form 1. The modifications of c_3, Int, and c_2 at lines 14, 12, and 15 of (33) lead to derivative code entries 1a, 1hh, and 1e respectively. These prederivative sequences are as follows,

(34) /* for the change to c_3 */

(∀z ∈ ([+: w ∈ ({a}-Int]E(w)-c_3) | z ∉ Int & c_2(z)=0)

c_4 := c_4 + {z};

end ∀;

(34') /* for c_2 */

if x ∈ c_3 & c_2(x) = [+: w∈{y}]1 & x∉Int *then*

c_4 := c_4 + {x};

endif;

(34") /* for Int */

(∀y ∈ ({a} - Int) | y ∈ c_3 & c_2(y) = 0)

c_4 := c_4 - {y};

end ∀;

Since uses of c_2 and c_3 occur within (34), (34'), and (34"), these values cannot be removed. The system will therefore request the user to supply names for c_2 and c_3. Finally, initialization code for c_4 will be inserted at the end of the prologue.

The FD command (29) will therefore initiate system actions taking the high level algorithm (28) into the following equivalent concrete version,

```
(35) 1       Int := {H};
             /* prologue */
     2       predout := nullset;  /* corresponds to c2 */
     3       (∀y ∈ V, x ∈ E(y) | y ∉ Int)
     4            if x ∈ Dom predout  then
     5               predout(x) := predout(x)+[+: w ∈ {y}]1;else
     6               predout(x) := [+: w ∈ {y}]1;
     7            endif;
     8       end ∀;
     9       succ := [+: w ∈ Int]E(w);  /* same as c3 */
     10      new := {x ∈ succ|x ∉ Int & predout(x) = 0};
             /* main loop */
     11      (while ∃ a ∈ new)
     12           (∀y ∈ ({a} - Int), x ∈ E(y) | y ∈ V)
     13                if x ∈ succ & predout(x) = [+: w ∈ {y}]1
                                   & x ∉ Int then
     14                     new := new + {x};
     15                endif;
     16                predout(x) := predout(x)-[+: w ∈ {y}]1;
     17           end ∀;
     18           (∀z ([+:w∈({a}-Int]E(w)-Succ)
                                   z ∉ Int & predout(z) = 0)
     19                new := new + {z};
     20           end ∀;
     21           succ := succ+([+: w∈({a}-Int]E(w)-succ);
     22           (∀y ∈ ({a}-Int) y ∈ succ & predout(y)=0)
     23                new :=  new - {y};
     24           end ∀;
     25           Int := Int + {a};
             end while;
```

Although (35) invites cleanup at almost every other line, (35) itself represents a considerable speedup over (27); i.e., we can expect (35) to run at worst in time proportional to #E.

As a last example of algorithmic improvement by formal differentiation, we consider Haberman's Banker's Algorithm for detecting deadlock amongst concurrent processes competing for resources (e.g., in an operating system environment). This algorithm models resource allocation by allocating 'loans' to customers who make known demands. Initially each customer c will have a quantity, loan (c), already preallocated to him, but he also requests claim (c) more money. Once his total demand is met, he will repay the bank his entire borrowed amount within a finite amount of time.

The bank starts out with an unallocated sum, cash, which it must use to satisfy the full demands of all of the customers one after the other.

The version of Haberman's algorithm given here uses only one kind of currency (equivalent to one resource type). Its strategy is to meet the demands of any customer c whose claim is less than the bank's available cash. The bank will then wait until c makes full repayment and is no longer a customer before scheduling any more customers. If all customers have been eliminated when the algorithm terminates, the original configuration of loans is *safe* (i.e., a deadlock can be avoided); otherwise not.

A base form SETL version of this algorithm is

```
(36)          /* cus is the set of customers */
     1        (while ∃ c ∈ cus | claim(c) ≤ cash)
     2             cash := cash + loan(c);
     3             cus := `cus - {c};
     4        end while;
```

To reduce (36) only one preparatory transformation is required. Specifically, P6 of Appendix D (v) should be applied to the existential quantifier within the *while* loop. The *while* loop predicate will then appear as

(37) $\exists\ c \in \{x \in cus \mid claim(x) \leq cash\}$

which is ready for reduction. A user can now issue the directive

(38) $FD,1, Gcus = {x ∈ cus|claim(x) ≤ cash}

to begin a rapid and dramatic transformation of (36).

Since IV_1 = {cus}, IV_3= {cash}, and RC= {claim, loan}, Gcus is reducible and matches elementary form 1 of F with variables cus, claim and cash matching the corresponding parameters x_1 , F 5 , and x_{11} of form 1. Of these parameters, only x_1 and x_{11} undergo change; cus is defined at line 3, cash at line 2, and these changes match parameter changes in D corresponding to derivative entries 1b and 1aa.

These entries lead to the following prederivative code,

```
            /* update code due to change in cus */
(39)        (∀x ∈ ({c} * cus)|claim(x) ≤ cash)
                 Gcus := Gcus - {x};
            end ∀;
```

and

```
(40)        /* due to change in cash */
            (while xmin_ll < cash + loan(c))
               (∀x ∈ {u ∈ cus|claim(u) = xmin_ll})
                  Gcus := Gcus +  x ;
               end ∀;
               xmin_ll := succ_ll(xmin_ll);
            endwhile;
```

The variables $xmin_{ll}$ and $succ_{ll}$ appearing in (40) must be initialized on entry to the loop. The initializing code for Gcus stems from entry 1 of Init and is given by the following code:

```
(41)        sortas(claim[cus], ll);
            xmin_ll := [min: w ∈ claim[cus]|w > cash]w;
            Gcus := {x ∈ cus|claim(c) ≤ cash};
```

where sortas sorts claim[cus] in ascending order and produces successor and predecessor maps, $succ_{ll}$ and $pred_{ll}$ for traversing this sorted set of numbers.

For FD to be profitable the derivative table entry lhh requires that the expression $c_1(xmin_{ll})=\{u\in cus|claim(u)=xmin_{ll}\}$

must be reducible. Hence, our FD system will define c_1 by inserting the following code,

(42)
```
c1 := nullset;
(∀x ∈ cus)
   if claim(x) ∈ Dom c1 then
      c1(claim(x)) := c1(claim(x)) + {x};
   else c1(claim(x)) := {x};
   endif;
end ∀;
```

at the end of the prologue. Just before line 3 of (36), the prederivative code

(43)
```
(∀x ∈ {c} * cus)
   c1(claim(x)) := c1(claim(x)) - {c};
end ∀;
```

obtained from entry lb of D will be inserted. Finally, $c_1(xmin_{11})$ will replace the calculation it represents within (40).

The version of the Banker's Algorithm which results from all this runs an order of magnitude faster than (36), and is as follows:

(44)

```
        /* prologue */
1       sortas(claim[cus], ll);
2       xmin_ll := [min: w ∈ claim[cus]|w > cash];
3       Gcus := {x ∈ cus|claim(c) ≤ cash};
4       goodc := nullset;     /*  goodc ≡ c_1 */
5       (∀x ∈ cus)
6            if claim(x) ∈ Dom goodc then
7                 goodc(claim(x)) := goodc(claim(x))+{x};
8            else goodc(claim(x)) := {x};
9            endif;
10      end ∀;
        /* main loop */
11      (while ∃c ∈ Gcus)
12           (while xmin_ll < cash + loan(c))
13                (∀x ∈ goodc(xmin_ll)
14                     Gcus := Gcus + {x};
15                end ∀;
16                xmin_ll := succ_ll(xmin_ll);
17           endwhile;
18           cash := cash + loan(c);
19           (∀x ∈ ({c} * cus)|claim(x) ≤ cash)
20                Gcus := Gcus - {x};
21           end ∀;
22           (∀x ∈ ({c} * cus))
23                goodc(claim(x)) := goodc(claim(x)) {x};
24           end ∀;
25           cus := cus - {c};
26      endwhile;
```

The above code requires some cleanup, but its asymptotic speed is still good -- at worst proportional to #cus log #cus. Note that the sort operation at line 1 may require this much time; however, when #claim[cus] << #cus, the expected running time of (44) will be O(#cus).

E. Conclusion

1. Implementation Plans

The two main goals proposed in this thesis for future work are the implementation of an interactive source to source Subsetl program manipulation system and the mechanization of formal differentiation for Subsetl. Since SETL provides machine portability, an interactive capability, readability, and a minimal programming effort we will use SETL to implement these projects.

The actual work can be completed incrementally in six phases. In the first three phases we plan to implement the basic source to source transformational system which also supports the FD design of Chapter 4 (C). This initial system will then be extended in the next three phases to make it useful for experimenting with and automating formal differentiation. Ultimately we hope to incorporate FD as part of an optimizing compiler.

A brief description of each phase of our proposed implementation project is as follows:

Phase 1.

First we plan to construct a basic interactive source to source Subsetl program improvement system as described in Chapter 3 (A,B). For this, we require the unparser, pattern matcher, and macro expander given in Appendix E (i-iii) and a transformational library containing several

of the set theoretic transformations found in Appendix D. In addition we must supply a Subsetl parser and a command processor which interfaces the transformational system and user.

Phase 2.

Next, we want to accomplish control flow, data flow, and type analysis. And once this is done, we can proceed to implement semiautomatic FD for completely continuous Subsetl expressions (cf., Chapter 3(C)).

Phase 3.

In this last preliminary phase, the simple pattern matching and macro expansion routines introduced in phase 1 should be replaced by the more powerful pattern handling procedures shown in Appendix E (v). Then we can implement the more ambitious semiautomatic FD implementation of Chapter 4 for general Subsetl expressions.

Phase 4.

Starting with the semiautomatic implementation completed in phase 3, we can now proceed to study program derivations which depend on formal differentiation so that we might gain sufficient empirical evidence to make FD fully automatic. To attain this goal, we must be able to mechanize three main tasks:

a. Reducible expressions must be recognized before preparatory transformations to produce them are even applied (cf., p. 470) for some initial ideas on how this

can be done).

b. Some of the reducible expressions recognized in the preceding step can then be selected for reduction. This will trigger application of a chain of preparatory transformations (using Kibler's chaining technique; cf., Appendix D (x)).

c. Preparatory transformations will lead seguet into successive reduction steps interleaved by applications of chains of cleanup transformations (once again, using Kibler's chaining technique).

There is little doubt that an effective mechanization of set theoretic FD will require declarations to supply important program facts which cannot be determined by automatic program analysis. For example, profitable differentiation of $c_2 = \{x \in s | q \in f(x)\}$ (cf., (29) of Chapter II (D)) depends on the property

(1) $\quad \# f(x) << \#s \quad$ for each $\quad x \in s$,

which would have to be declared in any practical implementation. Of course, when f is the successor or predecessor map defined on s, then any declaration stating that (f,s) represents a tree, a dag, or a flow graph would lead to the expectation that condition (1) holds. Note that differentiation of the topological sort and transitive closure algorithms in the last section requires condition (1).

We encounter another example of a program property which must be declared when choosing between two competing rules for reducing

(2) $$c_1 = \{x \in s \mid f(x) < q\} .$$

To differentiate (2) relative to incremental changes in q, one reduction method is preferred when the image set of integers f[s] is 'dense', while another method works much better when f[s] is 'sparse' (cf., (27) of Chapter II (D)). The latter method is used to differentiate the Bankers' Algorithms in Chapter 4 (D) and Appendix F.

Once we work out declarations for specifying such properties as sparsity, and once these declarations are sufficient to enable a fully automatic implementation of FD, then we will study how to incorporate automatic FD as part of the SETL compiler. FD is a higher level optimization and naturally precedes the techniques of automatic data structure selection which have recently been added to the SETL optimizer.

Note finally, that the success of installing FD as part of the SETL compiler may ultimately rest on the efficiency of the implementation. Thus, it may be necessary to improve our pattern handling techniques and to explore inexpensive ways to perform data flow and type analysis incrementally.

Phase 5.

Assuming that FD has been successfully automated in phase 4, we can proceed to expand the interactive system so that it can manipulate source programs written in full standard SETL. Next we will enlarge our initial collection of set theoretic FD transformations, and will explore new techniques for reducing a more general class of expressions.

Among those FD transformations which seem like promising candidates for inclusion into our system are the various incarnations of Rule 2. We can also admit FD table entries for two new reducible expression forms roughly corresponding to the set formers

(3) $\{x \in F_1(q_1,\ldots,q_n) \mid K_1(x,t_1,\ldots,t_m)$ *or* $\neg K_2(x,b_1,\ldots,b_\ell)\}$

based on (49) of Chapter II (D) and

(4) $\{K_3(x)\colon x \in F_2(t_1,\ldots,t_m)\}$

related to (56) of Chapter II (D). Note that within (3) and (4) the parameters q_i , i = 1,...,n, t_i , i = 1,...,m, and b_i , i = 1,...,ℓ, are discontinuities in which n,m,$\ell \geq 0$; F_1 must be a region constant; F_2 must belong to the induction set IV_4 (cf., p. 456); K_1 is restricted in the same way as the parameter K appearing in Forml of the F table in Appendix C (iii); K_2 is of the same form as K_1 but every free variable occurring within K_2 but outside of any b_i, i = 1,...,ℓ must be a region constant;

K_3 must be constrained by the condition Restrict(K,f) (cf., p. 427). We can further broaden the class of reducible expressions and useful FD transformations by adding appropriate FD entries for handling several of the examples discussed in Appendix F.

It may be somewhat more difficult but still worthwhile to study reduction techniques for handling discontinuous expressions some of whose discontinuity parameters require range analysis. An example of such an expression is

(5) $$c_1(y) = \{z \in y \mid z \notin s\}$$

which must be reduced in order to improve the grammar algorithm (6) of Appendix F. To reduce (5) we determine the range of values D_y of the discontinuity y by interrogating the Usetodef map. However, in more complicated situations range analysis may require comprehensive value flow (Sch8). In still more difficult situations it may be easier to fall back on range declarations.

We can also handle general discontinuities (which may not be removable) using the 'memo' function technique discussed in Chapters 1 and 2. Using this method, we are able to reduce all applicative expressions, but accurate speedup estimates are not generally possible since they are likely to depend on undecidable facts; i.e., improvement in running time will depend strongly on a large loop iteration and on small ranges of values for the discontinuities.

As is noted in Appendix F (cf., (28-30), (40)), it can be useful to develop general techniques to eliminate redundancy occurring among the discontinuities of an expression. For example, straightforward prederivative code for the setformer

(6) $c_1(q_1,q_2,q_1) = \{x \in s \mid q_1 \in K_1(x) \ \& \ q_2 \in K_2(x) \ \& \ q_1 \in K_3(x)\}$

relative to a change $s := s + \Delta$ is

(7) $(\forall x \in (\Delta - s),\ t_1 \in K_1(x),\ t_2 \in K_2(x),\ t_3 \in K_3(x))$

If $c_1(t_1,t_2,t_3) \neq \Omega$ *then*

$c_1(t_1,t_2,t_3)$ *with* x *else*

$c_1(t_1,t_2,t_3) := \{x\}$;

endif;

end $\forall$;

But by exploiting the redundant use of q_1 within (6), we can rewrite (7) using the following more efficient code:

(8) $(\forall x \in (\Delta - s),\ t_1 \in K_1(x),\ t_2 \in K_2(x) \mid t_1 \in K_3(x))$

If $c_1(t_1,t_2,t_1) \neq \Omega$ *then*

$c_1(t_1,t_2,t_1)$ *with* x *else*

$c_1(t_1,t_2,t_1) := \{x\}$;

endif;

end $\forall$;

In fact we can reform c_1 as a bivariate map by eliminating one of the q_1 parameters.

The preceding example leads to a general rule which

sometimes may be applied to handle redundant discontinuities. Consider an expression

(9) $c(x_{k+1},\ldots,x_n) = f(x_1,\ldots,x_K, x_{K+1}, \ldots, x_n)$

which depends continuously on modifications in x_i , i=1,...K and discontinuously on the remaining parameters. Suppose that the prederivative code for (9) which compensates for a change $x_i := \Delta_{x_i}$, $i \leq k$ is of the following form:

(10) $(\forall [t_{k+1},\ldots,t_n] \in \text{Project}(n-k,c))$
$c(t_{k+1},\ldots,t_n) := \delta c(t_{k+1},\ldots,t_n)$;
end ∀;

In order to handle the following new expression

(11) $c_1(x) = f(x_1,\ldots,x_k,x,x,\ldots,x)$

formed from (9) by replacing each discontinuity parameter of (9) by x, we note first of all that the map c_1 which stores separate calculations of (10) only uses a single parameter corresponding to x. We also observe that the prederivative code for (11) relative to a change $x_i := \Delta_{x_i}$, $i \leq k$, may be derived rather easily from (10); i.e., this update code is

(12) $(\forall t \in \textit{Dom}\ c_1 | [t,t,\ldots,t] \in \text{Project}(n-k+1, c_1\{t\}))$
$c_1(t) := \delta' c_1(t)$;
end∀;

where the expression $\delta'_{c_1(t)}$ appearing in (12) can be constructed from the expression $\delta_{c(t_{k+1},\ldots,t_n)}$ appearing in (10) using the following substitution steps:

1. First, we replace occurrences of $c(t_{k+1},\ldots,t_n)$ within $\delta_{c(t_{k+1},\ldots,t_n)}$ by occurrences of $c_1(t)$.
2. Then we replace all remaining occurrences of $t_{k+1},\ldots,t_n$ by occurrences of t.

Techniques which eliminate redundant discontinuities not only help to improve reduction of differentiable expressions, but also allow us to reduce expressions not normally suited for FD. For example, although the setformer

$$c_1(q_1,\ldots,q_n) = \left\{x \in s \mid \underset{i=1}{\overset{n}{or}}\ K_i(x) = q_i\right\}$$

is not reducible, the related setformer

$$(13)\quad c_2(q) = \{x \in s \mid \underset{i=1}{\overset{n}{or}}\ K_i(x) = q\}$$

can be reduced. To see this, note that (13) can be transformed into the following equivalent form,

$$(14)\quad c_2(q) = \{x \in s \mid [+: y \in \{1 \leq i \leq n \mid K_i(x) = q\}]1 \neq 0\}$$

The innermost subexpression $\{1 \leq i \leq n \mid K_i(x) = q\}$ within (14) may be reduced in a manner similar to (38) of Chapter 2 (D). It is clear how to reduce the other subexpressions.

Phase 6.

In this final experimental phase, we plan to uncover and then implement useful very high level dictions optimizable by FD. One example is Schwartz's 'Pursue Block' (Schl1, Schl2),

```
                ∀∀
(15)               Block
                end ∀∀;
```

which causes *Block* to be executed repeatedly until such execution results in no change in value to any variable. In addition to the form (15) Schwartz also allows pursue blocks to involve bound variables, e.g.,

```
                (∀∀x1∈s1,...,xn∈sn(x1,...,xn)|K(x1,...,xn))
(16)               Block(x1,...,xn)
                end ∀∀;
```

which does the same thing as

```
(17)  (while ∃x1∈s1 ,..., xn∈sn(x1,...,xn)|K(x1,...,xn) &
          execution of Block would result in a change of state)
                Block(x1,...,xn)
      end while;
```

but much more concisely.

Certainly, the inefficiency of the pursue block (16) can be a high price to pay for clarity. However, for situations when formal differentiation can improve (17), this cost may be avoided. Note in particular that the base form transitive closure algorithm of Chapter 4 (D) (cf., (14)) can be written more simply as

(18)
```
(∀∀a ∈ s)
     s := s + f{a};
end ∀∀;
```

and the main loop of the base form available expressions algorithm found in Appendix F (cf., (36)) can be crisply stated using the following pursue block,

(19)
```
(∀∀n ∈ N_F)
   AE(n) := [*: y ∈ pred(n)]((AE(y)*PR(y))+XE(y));
end ∀∀;
```

The preceding examples, (18) and (19) represent two of the four main cases in which Pursue Blocks of the form

(20)
```
(∀∀x ∈ s|K(x))
     statement
end ∀∀;
```

may be reduced. These four cases correspond to the following four 'monotonic' forms for *statement*:

(21)
```
A := A ± exp(x)
A(x) := A(x) ±  exp(x)
```

where exp(x) is any expression involving x.

To illustrate one of these cases, consider the Pursue Block

(22)
```
(∀∀x ∈ s|K(x))
     A(x) := A(x) + exp(x);
end ∀∀;
```

which can be implemented by the following lower level code,

```
(23)  (while ∃x ∈ s|K(x) & ((exp(x)-A(x)) ≠ nullset))
          A(x) := A(x) + exp(x);
      end while;
```

Without much ado (23) can be rewritten in the following form

```
(24)  (while ∃x ∈ {y ∈ s|K(y) & [+: ω ∈ {z∈exp(y)|z∉A(y)}]1≠0})
          A(x) := A(x) + exp(x);
      end while;
```

suitable for differentiation. The other three cases are handled in much the same way.

One long range implementation goal is to extend formal differentiation so that it can apply to full SETL procedures. However, before we are able to recognize the continuity properties of full procedures and to differentiate them automatically, it may be useful to augment SETL with declarations which state a procedure's continuity properties and associated derivative rules. This extended FD capability would facilitate the construction of large modular incremental programs; e.g., incremental metaparsers continuous relative to slight grammatical changes or incremental data flow algorithms continuous with respect to changes in local maps and the control flow graph.

ii. Various Implications and Applications of Formal Differentiation

There are many interesting side issues which can be pursued in parallel with the main implementation goals just mentioned. Some of these are listed as follows:

1. The most general algorithms presented in this thesis for recognizing differentiable expressions and performing the actual reductions have been derived from the heuristic notion of 'continuity', and can be used for any procedural programming language. To implement FD for a language P, we must select either the automatic or semiautomatic FD routines described in Chapters 3 (C) and 4(B), and must also construct the FD tables which encode the continuity properties of some of the primitive operations of P. Thus, it seems both feasible and worthwhile to design FD implementations for various high level languages such as Snobol and APL.

2. It is somewhat more speculative but still interesting to apply FD in a simplified data base context, where we neglect issues of sharing, and limit storage to main memory. Our basic idea is to make SETL an incrementally compiling and interactive language so that directives containing SETL source code to construct, modify, and query a data base can be issued from a terminal. These directives will have the form,

(25) op, setlsource

where op is an operation code, either 'construct', 'query', or 'modify', and setlsource is a block of SETL source state-

ments restricted in the following way: 1. Setlsource for 'construct' operations contains statements which assign initial values to certain 'elementary' variables of the data base. 2. Queries are SETL code sequences which use these 'elementary' variables (without modifying them) to form expressions which can be printed. 2. Directives having an operation code 'modify' involve SETL code which change 'elementary' variables differentially; i.e., in a way ensuring that these variables are 'inductive'.

A finite sequence of SETL source statements laid out from consecutive directives may be seen as forming a rather stylized straight line SETL program P having a characteristically high degree of repetitive code (as would normally be found only in a program loop). Thus, we can anticipate opportunities to improve P by application of various peephole transformations and the more global techniques -- redundant code elimination, formal differentiation, and data structure selection.

Unfortunately, program optimization methods depend on the availability and analysis of complete programs. And in the interactive milieu in which our data base model resides, only a part P' of our 'program' P is available for analysis -- the part which has already been executed. There always remains a significant unknown portion P" of P formed from directives yet to be issued. Nevertheless, we expect the

code of P to be sufficiently repetitive that properties of P" (especially some initial segment of P") can be predicted from analysis of P' (especially some final portion of P'). Consequently, it seems plausible that various program optimization techniques, reformulated in a minor way for run time use, can improve the processing of our data base directives.

In particular, we will show how formal differentiation might be used to optimize queries. Consider, as an example, a data base used by an airline company. Suppose that the 'elementary variables' of this data base are

1. A set Flights of flight numbers;
2. A map Strt associating each flight n ∈ Flights with a starting location Strt(n);
3. A map dest associating each flight n ∈ Flights with a destination dest(n).
4. A map Pass associating each flight n with a set Pass(n) of passengers scheduled to fly on flight n.
5. A map Food associating each flight n and each passenger p ∈ Pass(n) with a meal selection Food(n,p).

Then we can initialize the data base using the following directive,

(26) 'construct', Read(Flights,Strt,dest,Pass, Food);

Once the Read statement within (26) is parsed, compiled, and executed, we can issue an assortment of queries which use the five elementary variables just defined. Some examples are

```
(27) 'Query',Print(#{f∈Flights|Strt(f)='New York'
                              & dest(f) = 'Paris'});
(28) 'Query',Print(∃p∈Pass(142)|Food(p,142)='Kosher');
```

and

```
(29) 'Query',
     Q := {f∈Flights|∃p∈Pass(f)|Food(p,f)='Fish'}
     (∀f ∈ Q)
          Print(f, {p ∈ Pass(f) | Food(p,f) = 'Fish'});
     end ∀;
```

To ensure that the five elementary variables are inductive, we only allow code within 'modify' directives to change the maps strt, dest, and food by indexed assignments; also, the set Flights and each set Pass(n) for a given flight n may only vary by differential set additions and deletions.

Most of the techniques needed to optimize queries have been worked out in previous sections of this thesis. However, we also require that additional techniques mentioned in phase 4 of the FD implementation proposed in the preceding subsection be available. Thus, we assume that FD is fully mechanized, and specifically that hidden reducible expressions can be recognized before they would be exposed by application of preparatory transformations.

We now illustrate how the queries (27)-(29) might be improved using formal differentiation. First consider the case of (27). Once the Print statement appearing in (27) is parsed, we will recognize two reducible expressions,

$$c_1(q_1,q_2) = \{f \in \text{Flights} \mid \text{strt}(f) = q_1 \ \& \ \text{dest}(f) = q_2\}$$

and

$$c_2(q_1,q_2) = [+: x \in c_1(q_1,q_2)]1$$

Note that we ignore the fact that 'New York' and 'Paris' appearing in (27) are region constants, and instead treat (27) as containing two removable discontinuities q_1 and q_2. This strategy is taken in anticipation of encountering other reducible expressions differing from c_1 only with respect to the bound variable f and constants used in place of the parameters q_1 and q_2. All such expressions would be kept available by reduction of c_1 (cf. the rules of (36) of Chapter 2 (D)). It is also practical to reduce c_1 under the assumption that all continuity variables Flights, Strt, and dest can be modified.

In order to select expressions for reduction efficaciously, we keep track of all 'nonsimilar' reducible expressions (cf., (22) of Chapter 4(B)) and their frequency of occurrence within some finite number of most recently executed queries. In the case of c_1 and c_2, we must decide for each of these expressions whether it is similar to an expression which has been encountered 'frequently'. If this is the case for c_1 and if c_1 is not yet reduced, we will then perform the initialization part of the FD transformation for c_1. If c_1 is similar to a reduced expression, we can replace the expression c_1 in the query parse tree by the appropriate map retrieval term. After examining c_2 in the same way, the

query (27) can be compiled and executed. As a final step, we eliminate the reduced forms of all those reduced expressions whose frequency of occurrence has become too low.

After c_1 and c_2 are reduced, directives which 'modify' Flights, strt, or dest, will trigger execution of derivative code to maintain c_1 and c_2.

It should be expected that heavy use of the airline data base just described will very quickly establish expressions c_1 and c_2 as occurring frequently enough to warrant reduction. There may be several other expressions which will require a longer period (to be discovered by the user community) before they stabilize in reduced form. At the outset, queries for such a data base are likely to be executed inefficiently. Eventually, however, the system can be expected to reach an equilibrium state in which the small number of commonly occurring expressions most useful in the formation of queries will be detected and reduced.

We also expect occasions when persistent occurrences of usually rare queries will trigger temporary query optimization. Consider the following scenario. Somewhat by chance, a query (28) is issued to inquire whether Kosher food must be prepared for flight 142. Such a query establishes uses of two reducible expressions,

$$c_3(q_1,q_2) = \{p \in \text{Pass}(q_1) \mid \text{Food}(p,q_1) = q_2\}$$

and

$$c_4(q_1,q_2) = [+: x \in c_3(q_1,q_2)]1$$

(cf., (28)-(30) of Appendix F for reduction of c_3). If at the same time there are reports that certain contaminated fish have been distributed to various flights, then the probable use of emergency queries such as (29) reinforced by occasional queries of the form (28) can initiate reduction of c_3 and c_4 (which are common to both queries). Of course, when the emergency subsides and uses of c_3 and c_4 become rare, the maps holding values of c_3 and c_4 will be eliminated.

3. Another area of research involves expanding the results of Chapter 2(D) where we make some initial estimates of expected speedup which can result from application of FD to SETL expressions. If algorithm speedup and space utilization can be prediced in advance of successive applications of formal differentiation, then we move closer to the point when FD can actually be installed safely as part of a conventional compiling system. Of course, results of this kind will also make FD more powerful by allowing us to make better choices between competing transformations.

Studies along these lines might also lead to a more powerful use of stepwise refinement to prove space and time requirements of an algorithm in addition to its correctness. For this purpose we would apply FD transformations to a base form algorithm to determine an asymptotic speed and space estimate. We would then apply data structure selection transformations to obtain closer estimates of

the constant factors involved in the analysis of the final efficient form of the algorithm.

4. The heuristic notion of 'continuity' frequently arises in algorithm design and analysis. It seems highly worthwhile to study this property further with regard to more data structures than those used in the runtime environment of SETL. This in turn will shed new light on the continuity properties of full algorithms (which must be implemented using data structures and associated operations). In [Sch 6] Schwartz elaborates on this thought,

> Something close to the heuristic notion of 'continuity'... seems to play an important role in algorithm design. In [Sch4] we note that programs will commonly be structued as nests of loops; many of these loops ... realize some set-theoretic expression $E = E(a)$ by applying a map $M = M_a$ repeatedly until E emerges as a fixed point of M. The efficiency of programs having this structure can often be improved by noting that within an 'outer' loop L_{out} which contains an 'inner' loop L_{in} producing the value $E(a)$, the parameters a of $E(a)$ are varied only slightly. An observation of this kind often allows one to restructure L_{in} for efficiency by calculating E using its available previous value, which calculation can of course be substantially more rapid than calculation of E 'from scratch' would be. [Moreover, a speedup of this kind may still be realized even when the parameters a do not vary 'slightly' within L_{out}, if they can be made to do so by restructuring L_{out}.]... This line of thought makes it clear that an algorithm for evaluating $E = E(a)$ will be of particular interest if it has good continuity properties. Suppose for example that $E(a)$ is calculated as the fixed point of a transformation M_a. There will in general be many transformations $M_a, M'_a, M''_a, \ldots$ all of which have the value $E(a)$ as fixed point; among these transformations [we] will often be particularly interested in those M_a for which the sequence $E(a), M_{\bar{a}}(E(a)), M_{\bar{a}}(E(a))), \ldots$ leads after comparatively few iterations to the fixed point $E(\bar{a})$ of $M_{\bar{a}}$ (where we assume that the parameter values a and $\bar{a}$ differ only slightly). This line of thought points up a

problem area in algorithmic analysis which has not yet been explored systematically.

It is instructive to consider one or two cases in which algorithms or data structures having useful properties of continuity are known or can be devised. First consider sorting, and the problem of maintaining the sorted form of a set s to which modifications are continually being made by addition and deletion. If there are n elements in s, the bubble sort will correct for an insertion or deletion in approximately n/2 steps. However, if the sorted form of s is kept as a balanced tree, one can connect for an insertion or deletion in log n steps.

Next consider the minimum *min* of a set s of integers. After an insertion s = s + {x} one can update *min* by executing

```
min = if x lt min then x else min;
```

and after a deletion s = s - {x} by executing

```
min = if x ne min then min else (sort s)(1).
```

Since in many situations the minimum of s will rarely be deleted, it will rarely be necessary in using this procedure to generate the sorted form of s. On the other hand, if the minimum of s is used in a process, as for example a selection sort, which invariably deletes the minimum, then one wants an algorithm which has good 'worst case' rather than good 'typical case' continuity properties. In such a situation, it is reasonable to arrange s as a vector v = *tree* s having the implicit tree property, i.e. v(n) < v(2*n) and v(n) < v(2* n+1). Then the minimum of s is necessarily v(1), i.e. can be expressed as (*tree* s)(1). Note that in approaching the quantity *min* s in this way, we have essentially factored the function *min* into the product of two functions, of which the first, *tree*, is continuous, while the second (indexing) can be performed rapidly.

In addition to Schwartz's observations above we note that sometimes it is necessary to restructure a program loop L containing uses of an expression e in order to exploit the continuity properties of e. As an example of this, consider a loop

(30)
$$(\forall x \in s)$$
$$\ldots$$
$$= \{y \in T \mid f(y) \leq x\}$$
$$\ldots$$
end ∀

containing repeated calculations of

(31) $$c = \{y \in T \mid f(y) \leq x\}$$

and no definitions to either f or T. Unfortunately (31) cannot be reduced within (30) since x is not an induction variable of c. Thus the net cost of computing c within (30) is O(#s × #T). Note, however, that by selecting x values from s in sorted order, we can make x inductive and can differentiate (30) using the method (28') of Chapter 2(D). The cost of ordering s is O(#S × log #S), while the expense of keeping c available within (30) after reduction is O(#T × log #T + #S) -- and this usually represents improvement.

5. Another topic relevant to formal differentiation concerns expressions which cannot be reduced profitably. We hypothesize that the lower bounds in running time of algorithms whose base form versions depend heavily on such expressions will be predictably high. Substantiation of this hypothesis might then lead to the discovery of more general relationships between the lower bounds in running

time of algorithms and the continuity properties of expressions used in their base form rubble versions.

Of course, the problems just mentioned are very difficult, since finding lower bounds for a particular algorithm, and determining the full continuity properties of a single expression e are often hard problems. Note that while establishing reducibility for e can be achieved merely by discovery of some profitable derivative code, determining that e cannot be reduced requires the much more difficult discovery of a proof that no profitable derivative code exists at all. Proofs of this sort are little understood, however, and the most we have attempted to do in this thesis is provide a modicum of insight by considering example expressions whose continuity properties may be determined in a reasonable way.

Among the various expressions examined for their continuity properties, there arise two main obstacles to speedup due to FD. The most common snag results from the high cumulative cost of performing inexpensive update operations for too many stored calculations of an expression involving discontinuities. Speedup may also be limited when reduction of an expression e_0 depends on reduction of a whole chain of auxiliary expressions $e_1,\ldots,e_n$ in which for $i = 1,\ldots,n$, e_i must be reduced to make reduction of e_{i-1} profitable. (Recall that this situation arose in the first formulation of Rule 2, cf., p. 301.)

Although a chain of this sort may be necessary to minimize the update operations directly involving e_0, the net cost of reducing too many expressions may be prohibitive. In this case, an alternative approach which minimizes the number of auxiliary expressions at the cost of performing some redundant or unnecessary operations is preferred. (Rule 2 was reformulated in this way on p. 304.)

The preceding ideas can be illustrated using a few relevant examples which have not been discussed in the previous chapters. Consider the set union

(32) $$c_1(x) = f(x) + t$$

which is executed repeatedly within a program loop L. Suppose that within L, f is a region constant, t varies only by slight set additions and deletions, and x is modified *ad lib*. Suppose also that f is the edge map for a directed graph having *Dom* f as its vertices, and t is a subset of these vertices. Since (32) is discontinuous relative to changes in x, formal differentiation requires that separate calculations of (32) be stored in a map $c_1(x)$ for each $x \in$ *Dom* f. This may be achieved by performing the following initialization code,

(33)
```
c1 := nullset;
(∀y ∈ Dom f)
   c1(y) := f(y) + t;
end ∀;
```

on entrance to L. Then within L, whenever t changes by

$t := t \pm \Delta$, we can execute the following prederivative code,

(34)
```
(∀w ∈ (Δ ⁻* t), y ∈ {u ∈ Dom f | w ∉ f(u)})
        c1(y) with less w;
end ∀;
```

Examination of (33) shows that the iteration count N of the loop L must be greater than $\#Dom\ f$ or else the original cost of the repeated calculation (32) will be less than (33). But ignoring the cost of the preprocessing code (33), we see that the expense of computing the prederivative code (34) is at least proportional to the cardinality of the set

(35) $$c_2(w) = \{u \in Dom\ f \mid w \notin f(u)\}$$

For FD to be worthwhile we require that the cost of computing (32) which is proportional to $\#f(x) + \#t$ must be significantly greater than the cost of (35) (which is $O(\#c_2(w))$. But this will only be true when the graph f is almost completely connected; i.e., for all $u \in Dom\ f$, $f(u) \approx Dom\ f$.

Suppose now that f is almost completely connected. Then for FD to be profitable we must reduce (35) which, fortunately, can only be done profitably when f is almost completely connected (cf., p. 340). To reduce (35) we need to perform the following initialization immediately after (33):

(36)
```
c2 := nullset;
(∀y ∈ Dom f)
    c2(y) := {u ∈ Dom f | y ∉ f(u)};
end ∀;
```

Note that the cost of computing (36) is proportional to $(Dom\ f)^2$. Consequently, we can replace the costly setformer $\{u \in Dom\ f \mid w \notin f(u)\}$ occurring within (34) by the retrieval $c_2(w)$.

It is interesting to consider how to handle (32) in the following more restricted context: Suppose that t is an upper bound on the range of x values within L and that t only varies by differential set deletions. To exploit this new situation, we can replace the initialization code (33) and (36) by the following more efficient code,

(37)
```
c1 := nullset;
(∀y ∈ t)
    c1(y) := c1(y) + t;
end ∀;
c2 := nullset;
(∀y ∈ t)
    c2(y) := {u ∈ t | y ∉ f(u)}
end ∀;
```

Within L whenever t is modified by a change t := t - Δ we can execute the prederivative

(38) $(\forall w \in (\Delta * t),\ y \in c_2(w))$

$c_1(y)$ *less* w;

end ∀;

which is less expensive to compute than (34).

Note, however, that within (38) $c_2(w)$ can contain elements belonging to $\Delta * t$, so that unnecessary update operations for c_1 may occur. To avoid executing these undesirable operations, we can perform the following prederivative code for c_2 ,

(39) $(\forall w \in (\Delta * t),\ y \in \{u \in t \mid u \notin f(w)\})$

$c_2(y)$ *less* w;

end ∀;

just prior to (38). But by introducing (39), an additional expression,

(40) $c_3(w) = \{u \in t \mid u \notin f(w)\}$

must be reduced.

To reduce (40) we perform the initialization

(41) c_3 := *nullset*;

$(\forall y \in t)$

$c_3(y) := \{x \in t \mid x \notin f(y)\}$;

end ∀;

immediately after (37). Consequently, the setformer $\{u \in t \mid u \notin f(w)\}$ occurring within (39) may be replaced by the retrieval $c_3(w)$. However, even after this, the

fact that c_3 depends on an ever decreasing set t implies that unnecessary update operations of c_2 can occur within (39) unless c_3 is first updated to correct for this problem. Unfortunately the required update calculation is

(42)
```
(∀w ∈ (Δ * t), y ∈ {u ∈ t | w ∉ f(u)})
        c3(y) less w;
    end ∀;
```

which can also include unnecessary updates of c_3.

Observe that a continued quest to eliminate unnecessary operations will only introduce more code selected from the cycle (38), (39), (42), and the cumulative expense of computing this code may be high. Thus it seems prudent to avoid this approach and instead be satisfied with the imperfect code (42) in which the setformer $\{u \in t \mid w \notin f(u)\}$ is replaced by the retrieval $c_2(w)$.

The preceding example suggests that workset algorithms can sometimes reach maximum efficiency by actually introducing local inefficiencies; e.g., by letting these algorithms depend on worksets which are larger than absolutely necessary.

iii. Summary

We have now presented a thesis investigating formal differentiation, a program optimization technique which generalizes and reformulates John Cocke's method of strength reduction, and provides a convenient framework with which to implement Jay Earley's 'iterator inversion'. Algorithms

to implement formal differentiation both automatically and semiautomatically for programming languages ranging from Fortran to SETL have been given. However, since FD seems best suited for SETL, we have studied set theoretic formal differentiation in depth, and have presented a comprehensive semiautomatic implementation design for a subset of SETL. This design includes an interactive source to source transformation program improvement system to be used for performing experiments in algorithm derivation using FD. It is expected that empirical evidence gathered from such experiments will lead eventually to a full mechanization of FD for SETL.

In contrast to other research in program transformations, we have shown that FD is unusual in many respects; e.g.,

1. It may be applied over a large spectrum of language levels and in wide ranging contexts within these languages.
2. It can realize swift convergence from a very high level inefficient form of an algorithm to a much lower and more efficient implementation version.
3. FD can be implemented systematically.
4. We can estimate the speedup due to transformations applied by our proposed SETL FD implementation, and have shown this speedup to be as great as an order of magnitude.

We have illustrated our proposed FD system for SETL by considering and improving eight sample Subsetl programs.

We feel that these initial case studies lend strong support to further efforts to fully automate and incorporate set theoretic formal differentiation as part of an optimizing compiler. There are encouraging indications that this goal will be reached. When this finally happens, it will represent real progress towards the development of optimization algorithms envisioned by Schwartz "which explore spaces of program transformations as freely as a manual programmer does" (Sch 6).

Appendix A. SETL and SUBSETL

In the present section, we summarize the principal features of a version of the SETL language used throughout this thesis. Much of this description has been taken from [Schl] pp. 72-80 and from [DGS 2]. However, some of the latest changes occurring in the official SETL language are absent, and we do not regard this *precis* as a reference to the current standard SETL language.

As the name suggests, SUBSETL is a subset of the version of SETL described in this section. We use an asterisk to the left of an item to denote a SETL feature not included in SUBSETL.

Basic Objects

Sets and *atoms*; sets may have atoms or sets as members. Atoms may be

	Integer	Examples: 0, 2, -3
*	Real	Examples: 9., 0.9, 0.9E-5
	Boolean	Examples: *True*, *False*
*	Character strings	Examples: 'aeiou', 'spaces'
*	Blank (created by function *newat*).	
	(Note: special undefined blank atom is Ω.)	

Basic Operations for Atoms

Integers: arithmetic: +, -, *, /, **, // (remainder)

comparison: =, ¬=, < , > , ≤ , ≥

(Results are *True*, *False*, or Ω.)

other: *max*, *min*, *abs*

Examples: 5//2 is 1; 3 *max* -1 is 3; *abs* -2 is 2.

*Reals: Above arithmetic operations (with exception of //) plus exponential, log, and trigonometric functions.

*Booleans: logical: &, *or*, *exor*, *implies*, ¬

(Any value other than *True* is considered *False*.)

logical constants: *True*, *False*

*Strings: + (catenation), * (repetition), a(i:j) (extraction), # (size), *nullchar* (empty strings).

Examples: 'a' + 'b' is 'ab'; 2 * 'ab' is 'abab', 'abc'(1:2) is 'ab', 'abc'(2:2) is 'bc', 'abc'(2) is 'b', # 'abc' is 3, # *nullchar* is 0.

General: Any two atoms may be compared using = or ¬=;

* *atom* a tests if a is an atom.

Basic Operations for Sets

∈, ∉ (membership tests); *nullset* (empty set), ∋ (arbitrary element), # (number of elements); = , ¬= (equality tests); *incs* (inclusion test); *with*, *less* (additionand deletion of element); pow(a) (set of all subsets of a);

* npow(k,a) (set of all subsets of a having exactly k elements)

Examples: a ∈ {a,b} is *True*, a ∈ *nullset* is *False* ∋ *nullset* is Ω; ∋ {a,b} is either a or b, #{a,b} is 2, # *nullset* is 0, {b} *with* a is {a,b}, {a,b} *less* a is {b}, {a,b} *less* c is {a,b}, {a,b} *incs* {a} is *true* pow({a,b}) is {*nullset*,{a},{b},{a,b}}. npow(2,{a,b,c}) is {{a,b},{a,c},{b,c}}.

Tuples

Ordered *tuples* are treated as SETL objects of different type than sets — e.g. tuples may have some components undefined.

Operations on tuples:

$\notin$, $\in$ (membership tests); *nulltuple* (empty tuple);

$\ni$ (arbitrary element).

Tuple former: If x,y,...,z are n SETL objects then

t = [x,y,...,z] is the n-tuple with the indicated components.

#t is the number of components of t

t(k) is the k-th component of t

t(i:j) is the tuple whose components, for $1 \leq k \leq j$, are

t(i+k-1)

+ is the concatenation operator for tuples

Examples: a(n) $\in$ t is an abbreviation for $\exists 1 \leq n \leq \#t \mid t(n)=a$

If t = [a,b] and τ = [a,c] then

T = t + τ = [a,b,a,c] , T(3:2) = [a,c]

Tuple components may be modified by writing t(j) = x;

An additional component may be concatenated to t

by writing t(#t + 1) = x;

Set Definition

By enumeration: {a,b,...,c}.

Set former: $\{e(x_1,\ldots,x_n): x_1 \in e_1, x_2 \in e_2(x_1), \ldots x_n \in e_n(x_1,..x_{n-1}) \mid c(x_1,\ldots,x_n)\}$.

Tuples may also be defined by analogous tuple-formers,

$[e(x_1,\ldots,x_n,m_1,\ldots,m_n): x_1(m_1) \in e_1,$

$\ldots, x_n(m_n) \in e_n(x_1,\ldots,x_{n-1},m_1,\ldots,m_{n-1})$

$\mid c(x_1,\ldots,x_n,m_1,\ldots,m_n)]$

The *range restrictions* x ∈ a(y) can have the alternate *numerical form*

$$\min(y) \leq x \leq \max(y)$$

when a(y) is an interval of integers.

If t is a tuple, the form x(n) ∈ t can be used, see below, *iteration headers*, for additional detail.

Optional forms include

{x ∈ a | C(x) equivalent to {x: x ∈ a | C(x)}; and

{e(x): x ∈ a} equivalent to {e(x): x ∈ a | *True*} .

Functional Application (of a set of ordered pairs, or a programmed, value-returning function):

f{a} is {if #p > 2 then p(2:) else p(2), p ∈ f |

if *type* p ¬= *tupl* then *False* else (#p) ≥ 2 & p(1) = a},

i.e. is the set of all x such that [a,x] ∈ f .

f(a) is: if #f{a} = 1 then ∋f{a} else Ω,

i.e., is the unique element of f{a}, or is undefined atom.

f[a] is the union over x ∈ a of the sets f{x},

i.e., the *image* of a under f.

More generally:

f(a,b) is g(b) and f{a,b} is g{b}, where g is f{a};

f[a,b] is the union over x ∈ a and y ∈ b of f{x,y}.

If f is a value-returning function, then

f{a,b} = {f(a,b)}, f[a] = {f(x): x ∈ a}, etc.

Constructions like f{a,[b],c}, etc. are also provided.

Compound Operator

$[op: x \in s]e(x)$ is $e(x_1)\ op\ e(x_2)\ op\ \ldots\ op\ e(x_n)$,

where s is $\{x_1,\ldots,x_n\}$.

This construction is also provided in the general form

$$[op: x_1 \in e_1,\ x_2 \in e_2(x_1),\ldots,x_n \in e_n(x_1,\ldots,x_{n-1}) \mid C(x_1,\ldots,x_n)]e(x) ,$$

where the range restrictions may also have the alternate numerical form, or the form appropriate for tuples.

Examples: $[max: x \in \{1,3,2\}](x+1)$ is 4,

$[+: x \in \{1,3,2\}](x+1)$ is 9,

$[+: x(n) \in a]x$ is SETL form of $\sum_{i=1}^{n} a$

$[op: x \in nullset]e(x)$ is Ω.

Quantified Boolean Expressions

$\exists x \in a \mid C(x)$ $\forall x \in a \mid C(x)$

General form is

$$\exists x_1 \in a_1,\ x_2 \in a_2(x_1),\ \forall x_3 \in a_3(x_1,x_2),\ldots \mid C(x_1,\ldots,x_n)$$

where the range restrictions may also have the alternate numerical form, or the form appropriate for tuples.

Evaluation of

$$\exists x \in a \mid C(x)$$

sets x to first value found such that C(x) = *True*.

If no such value, x becomes Ω.

The alternate forms:

$\exists min \leq x \leq max$, $\exists max \geq x \geq min$, $\exists max \geq x > min$, $x(n) \in t$, etc.

of range restrictions may be used to control order of search.

Conditional Expressions

if $bool_1$ then $expn_1$ else if $bool_2$ then $expn_2$ ... else $expn_n$.

Statements: are punctuated with semicolons.

Assignment and Multiple Assignment Statements

a := expn; f{exp} := expn; is the same as

f := {p ∈ f | p(1) ¬= exp} + {[exp,x]: x ∈ expn};

f(exp) := expn; is the same as f{exp} := {expn};

f(a,b) := expn; f{a,b} := expn; etc. also are provided.

* [a,b] := expn; is the same as a := expn(1); b:= expn(2:);

* [a,b,...,c] := expn; [a,[b,c],...,d] := expn; etc.

are also provided.

* [f(a), g{b}] := expn; is the same as f(a) := expn(1);

g{b} := expn(2:);

Generalized forms:

* [f(a), g{b,c},...,h(d)] := expn;

* [f(a), [g{b,c},h(d)], ...,k(e)] := expn; etc. also are

provided.

* Use of General Expressions on Left-Hand Side of Assignment Statements (sinister calls).

$e(x_1,...,x_n)$:= y; must be no-op if executed immediately after y := $e(x_1,...,x_n)$; and vice-versa. The use

op op' x := y;

of a product operator on the left-hand side of an assignment expands as

```
t := op' x;
op t := y;
op' x := t;
```

with sinister rules for multiparameter compounding. These rules allow user-defined functions to be used quite generally on the left-hand side of assignment statements. The 'left hand' significance of a function is often deducible from its standard right-hand side form, but may be varied by using specially designated code blocks which are executed only if the function is called from right-hand or left-hand position respectively. These have the respective forms:

(load) block; (execution only if function called from right-hand side is assignment)

(store x) block; (execution only if function f called is from $f(param_1,...,param_n)$:= x;) .

* Commonly Used Operators Having Special Side Effects

x := expn; has same value as expn and assigns this value to x

x *in* s; same as s := s *with* x;

x *from* s; same as x := ∋s; s := s *less* x;

x *out* s; same as s := s *less* x;

* Use of Code Blocks Within Expressions

If *block* is a section of text which could be the body of a function definition, then [; block] is a valid expression which both defines and calls this function. Such code block expressions can be used freely within other expressions.

Control Statements

* go to label;

if $cond_1$ then $block_1$ else if $cond_2$ then $block_2$...else $block_n$;

if $cond_1$ then $block_1$ else ... else if $cond_n$ then $block_n$;

Iteration Headers

(while cond) block;

* (while cond doing blocka) block;

$(\forall x_i \in a_1,\ x_2 \in a_2(x_1),\ldots,x_n \in a_n(x_1,\ldots,x_{n-1}) \mid C(x_1,\ldots,x_n))$ block;

in this last form, the range restriction may have such alternate numerical forms as

$min \leq x \leq max$, $max \geq x \geq min$, $min \leq x < max$, etc.,

which control the iteration order.

If t is a tuple, then the operator of form

(∀x(n) ∈ t) block; is available. This is abbreviation for

$(\forall 1 \leq n \leq \#t \mid t(n) \neq \Omega)$ x = t(n); block;

Iterators of this form may also be used in set formers, compound operators, quantifiers, etc.

Iterator Scopes

The scope of an iteration or of an *else* or *then* block may be indicated either with a semicolon, with parentheses, or in one of the following forms:

end ∀; end while; end else; end if; etc.,

or:

end ∀x; end while x; end if x; etc.

* Loop Control

quit; quit ∀x; quit while; quit while x;

and

continue; continue ∀x; continue while; continue while x;

The *quit* statement terminates an iteration; the *continue* statement begins the next cycle of an iteration.

* Subroutines and Functions (are always recursive)

To Call Subroutine:

sub(param$_1$,...,param$_n$);

sub[a]; is equivalent to (∀x ∈ a) sub(x);;

Generalized Forms:

sub(param$_1$,[param$_2$,param$_3$],...,param$_k$)

are also provided.

* To Define Subroutines and Functions

Subroutine:

define sub(a,b,c); text end sub;

return; — used for subroutine return

Function:

definef fun(a,b,c); text end fun;

return val; — used for function return

Infix and prefix forms:

define a *infsub* b; text end *infsub*;

definef a *infin* b; text end *infin*;

define *prefsub* a; text end *prefsub*;

definef *prefun* a; text end *prefun*;

* Namescopes

Scope declarations divide a SETL text into a nested collection of scopes. Scope names are known in immediately adjacent, containing, and contained scopes. Other than this, names are local to the scope in which they occur, unless propagated by *include* or *global* statements.

Declaration forms

scope name; ..., end name;

scopes with specified numerical level

scope n name; ..., end name;

global declaration

global $name_1$, ..., $name_n$;

with specified numerical level

global n $name_1$, ..., $name_n$;

include statement

include $list_1$, ..., $list_n$

Example:

* *include* bigscope1(scope1 x,scope2(-z),scope3(x,y,u[v])),
bigscope2*;

'*' signifies all elements in scope,

'-' signifies exclusion of those elements listed,

[ ] modifies the 'alias' under which an element is known in scope in which included. Subroutines and functions are scopes of level 0. Macros (see below) are transmitted between scopes in much the same way as variable names. The declaration

owns $routname_1(x_1,\ldots,x_{n1})$, $routname_2(y_1,\ldots,y_{n2})$, ...

states that the variables x_j are stacked when $routname_1$ is entered recursively, the variables y_j are stacked when $routname_2$ is entered recursively, etc.

* Macro Blocks

To define a block: *macro* mac(a,b); text *endm* mac;

To use: mac(c,d);

* Initialization

initially block; (*block* executed only first time process entered)

* Input-Output

Unformatted Character String:

A SETL file is a pair [st,n], where st is a character string and n an integer designating 1 of its characters. *er* is end record character; *input*, *output* are standard I/O media; the function record(s); — reads a file [st,n]

from position n until *er* character or string-end is encountered in the character string *st*.

* Standard Format I/O

f *read* $name_1,...,name_n$; using standard format reads from file [st,n], starting at position n

f *print* $expn_1,...,expn_n$; using standard form transfers external representation of objects to file s = [st,n], starting at position n as above.

The set $\{s_1,...,s_n\}$ is represented as $\{r_1,...,r_n\}$, where r_j is the external representation of s_j. Similarly, the tuple $[s_1,...,s_n]$ is represented as $[r_1,...,r_n]$.

If str is a character string identical with the external name under which a file is known to the operating system supporting SETL, then

open str returns a pair [st,1] ,

where *st* is the contents of the file *str*.

close(st,str)

makes the SETL string *st* into the contents of the external file *str*.

The following grammar for SUBSETL is a slight modification and subset of the SETLA grammar found in *SETL Newsletter* # 70, pp. 31-34. The notations used are the same as in that newsletter and are as follows:

<Stype>	Denotes a syntactic type
<*Ltype>	Denotes a lexical type
Literal, 'Literal'	Denotes literals
<Stype*>	Denotes indefinitely many repetitions of a syntactic type
<Stype(M,N)>	Denotes a minimum of M and a maximum of N repetitions of a syntactic type.

Lexical Types:

<*Name>	variable name
<*Opname>	operator (+,-,*,/,*max*,*min*,...)
<*Qname>	period terminating constant (*True*,*False*,*nullset*,*nullchar*, *nulltuple* ...)
<*Logop>	Logical operators (>, =, ¬=, ≥, *Incs*, ...)

SUBSETL GRAMMAR

```
<program> = <declaration *> <Block>
<declaration> = <type>(<varlist>);
<varlist> = <*name> <comname*>
<comname> = ',' <* name>
<type> = integer | boolean | tuple | set

<block> = <statement(1,*)>
<statement> = if <expn> then <block> <elseif*> else
                  <block> endif;
            = if <expn> then <block> <elseif*>  endif;
            = ∀<iterator> <block> end ∀;
            = (while <expn>)<block> end while;
            = <term> ':=' <expn>;
<term>      = <*name>
            = <*name> <arglist>
<arglist>   = (expnlist)
            = {expnlist}
<expnlist>  = <expn> <comexpn*>
<compexpn>  = ',' <expn>
<elseif>    = elseif <expn> then <block>
<iterator>  = <firstit(0,1)><iterlist><lastit(0,1)>
<firstit>   = <expn>','
<lastit>    = |<expn>
<iterlist>  = <iterexpn> <comiterexpn*>
```

```
<iterexpn>   = <*name> ∈ <expn>
             = <expn><compareop><*name><compareop><expn>
<compareop>  = >
             = '='
             = <  '='
             = <
             = > '='
             = '¬='
<comiterexpn> = ',' <iterexpn>
<expn>       = <factor><*opname><expn>
             = <factor>
             = if <expn> then <expn><elsexpn *> else <expn>
<elsexpn>    = else if <expn> then <expn>
<factor>     = <*opname><factor>
             = ∃ <iterlist> <lastit(0,1)>
             = ∀ <iterlist> <lastit(0,1)>
             = [<*opname>: <iterlist> <lastit(0,1)>]<expn>
             = <atom> <arglist>
             = <atom>
             = <atom> <*logop> <factor>
<ATOM>       = <term>
             = (<expn>)
             = {<expnlist>}
             = {<iterator>}
             = [<expnlist>]
             = [<iterator>]
             =  <*const>
             =  <*Qname>
```

Appendix B. Predicting Speedup for Rule 1

In the following Appendix we will continue the discussion raised in Chapter 2 (C) concerning the difficulties in predicting program speedup which results from formal differentiation. To illustrate our point we will consider the rule 1 reduction transformation applied to the setformer

$$c = \{x \in s \mid K(x)\} \tag{1}$$

executed repeatedly in a program loop L (cf. Chapter 2(C)).

In this case we can facilitate prediction of program speedup by restricting our attention to an asymptotic speed complexity in which the frequency of execution of a program loop L is arbitrarily large in comparison to the initialization block to L. This optimistic heuristic obviates the need to consider the cost of any code inserted within an initialization block to L by rule 1. Another possibliity is to take the somewhat pessimistic view that within L control will always travel along a path containing a maximal number n of prederivatives $c = c \pm \{x \in \Delta_1 \mid K(x)\}$, ..., $c = c \pm \{x \in \Delta_n \mid K(x)\}$ between any two calculations of (1). Then, after waving aside numerous complications, we can formalize a condition under which we expect rule 1 to result in improvement: For the cost of calculating (1) to exceed the maximum cost of computing the update code in L, the following inequality

$$(2) \qquad \sum_{i=1}^{n} \#\Delta_i \; Cost(K) < \#s \; Cost(K)$$

which simplifies to an equivalent 'win predicate'

$$(2') \qquad \sum_{i=1}^{n} \#\Delta_i < \#s$$

must hold.

The present thesis will not attempt to provide a method for static determination of (2'). Although we can factor out the term Cost(K) from the win predicate for the rule 1 case, a more general complexity measure cannot be avoided in analyzing more complicated examples of interest. Hence, the current thesis will steer clear of such serious and difficult complexity issues to favor a more heuristic approach having broad practical applications.

APPENDIX C

FORMAL DIFFERENTIATION TABLES

In this appendix we provide the pattern tables which support the various formal differentiation implementation designs discussed in Chapters 3 and 4. In particular, Section (i) contains the elementary form table (F) and the derivative table (D) used in connection with the Fortran variants of Algorithm 1-2, Algorithm 1, and Algorithm 2 (cf., Chapter 3 (C.2.3, C.3.2)). Section (ii) gives the F and D tables needed for our algorithms implementing set theoretic FD limited to expressions continuous in all of their parameters (cf., Chapter 3 (C.2.3, C.3.2, C.3.3)). Finally Section (iii) provides the F, D, Init, and Replace tables to be included as part of the more general SETL FD implementation design proposed in Chapter 4.

The entries in all of the tables to be presented consist of pattern specifications used for either pattern matching or macro expansion operations. To specify pattern expressions we will make use of literal symbols, pattern variables, pattern names, and names of code procedures, all combined by balanced parentheses and operations of concatenation, alternation, and predecessor formation. However, to make these tables more readable we will avoid cluttering up patterns with explicit tree structure, and

will instead assume that this structure can be determined by an appropriately modified Parser. Other notational shortcuts taken for the sake of clarity will be mentioned as we proceed.

The following BNF rules describe the notational details of our nonprocedural pattern language:

```
<ASSIGN> ::= <Pattern name*> '=' <Pattern expression>
<Pattern expression> ::= <term> |
                         <Pattern expression> '|' <term>
<term> ::= <factor> |
           <term> <factor>
<factor> ::= <literal*> |
             <Pattern variable*> |
             <Pattern mame*> |
             <Procedure name*> |
             [<Pattern expression>] |
             (<Pattern expression>)
```

The lexical categories of the above grammar are defined as follows:

<Pattern name*> -- an alphanumeric string beginning with a letter; each pattern name must appear once on the left side of an assignment.

<Literal*> -- a string enclosed in quotes;

<Pattern variable*> -- an alphanumeric string beginning with a letter; these identifiers can be distinguished from pattern names in that they cannot appear on the

left side of an assignment.

<Procedure name*> -- an alphanumeric string beginning with the symbol !;

i. FD Tables for Fortran

The following tables shown in abbreviated form can be used in connection with Algorithm 1-2 of Chapter 3(C,2.2) tailored to Fortran (cf., Chapter 3 (C, 2.3)). For Fortran FD only two tables, F and D, are required. Each of the three patterns f belonging to F appears as part of a term E = f. We use this notation to indicate that the pattern variable E will match the generated variable $v_{\bar{f}}$ used to hold the value of a reduced expression $\bar{f}$ matched by f.

The D table is lined up with the F table so that derivative entries associated with each elementary form f are listed just to the right of the entry for f. As a notational shortcut, we sometimes specify two derivative entries on a single line; e.g., the line

x1 = $\pm$ x3 E = $\pm$ x3 * x2

indicates the following two entries:

x1 = + x3 E = + x3 * x2

and

x1 = - x3 E = - x3 * x2

Elementary Form Table (F)	Derivative Table (D)	
	Parameter Change	Prederivative
1. E = x1 * x2	x1 = ± x3	E = ± x3 * x2
	x1 = - x3±x4	E = - x3*x2 ± x4*x2
	x1 = x3±x4	E = x3*x2 ± x4*x2
	x2 = ± x3	E = ± x3 * x1
	x2 = - x3±x4	E = - x3*x1 ± x4*x1
	x2 = x3±x4	E = x3*x1 ± x4*x1
2. E = x1 / x2	x1 = ± x3	E = ± x3/x2
	x1 = - x3±x4	E = - x3/x2 ± x4/x2
	x1 = x3±x4	E = x3/x2 ± x4/x2
3. E = x1 ** x2	x2 = ± x3	E = x1**±x3
	x2 = - x3±x4	E = (x1**-x3)*(x1**±x4)
	x2 = x3±x4	E = (x1**x3)*(x1**±x4)

ii. FD Tables for Subsetl Expressions Continuous in All of Their Parameters

The F and D tables shown below can be used with the Subsetl variants of either Algorithm 1.2 (cf., Chapter 3 (C.2.3)) or Algorithm 1 and Algorithm 2 (cf. Chapter 3 (C.3.2, C.3.3)). The form of these tables is the same as the tables just given for Fortran.

F Table	Parameter Change	D Table Prederivative
1. E = xl+x2	xl := xl + Δ;	E := E + Δ;
set union	xl := xl - Δ;	E := E - (Δ - x2);
	x2 := x2 + Δ;	E := E + Δ;
	x2 := x2 - Δ;	E := E - (Δ - xl);
2. E = xl*x2	xl := xl ± Δ;	E := E ± Δ * x2;
intersection	x2 := x2 ± Δ;	E := E ± Δ * xl;
3. E = xl-x2	xl := xl ± Δ;	E := E ± (Δ - x2);
set difference	x2 := x2 + Δ;	E := E - (Δ * xl);
	x2 := x2 - Δ;	E := E + (Δ * x2);
4. E = *Pow*(xl)	xl := xl + Δ;	E := E+ {y+z:y∈E,z∈*Pow*(Δ-x)};
	xl := xl - Δ;	E := E- {y∈E|Δ*y¬=*Nullset*};
5. E={x∈xl|k}	xl := xl ± Δ;	E := E ± {x ∈ Δ | k};
	f_k(yl,...,yn):=z;	s_0:= {x∈xl| *or*($\&_{i=1}^{r}$ $\&_{j=1}^{n}$ $P_{ij}(x)=y_j$)};
		E := E - {x ∈ s_0 | k};
postderivative ——————→		{E := E + {x ∈ s_0 | k};
6. E=[*op*:x∈xl|kl]k2	xl := xl+Δ;	E := E *op*[*op*:x∈(Δ-xl)|kl]k2;
where *op* is any binary operator		
7. E=[+:x∈xl|kl]k2	xl := xl-Δ;	E := E-[+:x∈(Δ*xl)|kl]k2;
where + is addition		

F Table	Parameter Change	Predederivative
8. E = {k1: x∈x1|k2}	x1 := x1+Δ;	E := E + {k1: x ∈ Δ | k2};

9. E = {x1≤x≤x2|k}	x1 := x1+Δ;	*if* Δ ≥ 0 *then*
		E := E-{x1≤x<(x1+Δ *min* x2)|k};
		else
		E:=E+{x1+Δ≤x<(x1 *min* x2)|k};
		endif;

	x1 := x1-Δ;	*if* Δ ≥ 0 *then*
		E:=E+{x1-Δ≤x<(x1 *min* x2)|k};
		else
		E:=E-{x1≤x<(x1-Δ *min* x2)|k};
		endif;

	x2 := x2+Δ;	*if* Δ ≥ 0 *then*
		E:=E+{(x1 *max* x2)<x≤x2+Δ|k};
		else
		E :=E-{(x1 *max* x2+Δ)<x≤x2|k};
		endif;

	x2 := x2-Δ;	*if* Δ ≥ 0 *then*
		E:=E-{(x1 *max* x2-Δ)<x≤x2|k};
		else
		E:=E+{(x1 *max* x2)<x≤x2-Δ|k};
		endif;

10. E=$[^{min}_{max}$:x∈x1|k1]k2	x1:=x1+Δ;	E := E $^{min}_{max}$ $[^{min}_{max}$: x∈Δ|k1]k2;

F Table	Parameter Change	Prederivative
11. E = x1 ∈ x2	x2 := x2 ± Δ;	*if* x1 ∈ Δ *then* E:= *true*; *false* *endif*;
12. E = x1 ∉ x2	x2 := x2 ± Δ;	*if* x1 ∈ Δ *then* E:= *false*; *true* *endif*;
13. E = x1 + x2	x2 := x2 + x3;	E := E + x3;
tuple concatenation	x1 := x3 + x1;	E := x3 + E;
	x2(x3) := x4;	E(#x1 + x3) := x4;
	x2(x3:x4) := x5;	E(#x1 + x3:x4) := x5;

iii. FD Tables for Subsetl Expressions Continuous in Some of Their Parameters

The F, Replace, Init, and D tables shown below support the SETL implementation design discussed in Chapter 4 (C). Because too many long and complicated pattern entries are required for these tables, it is no longer convenient to present them using the simple format for tables given in Sections i and ii. Instead, they are given separately with the following rule of association: Within each table, entries corresponding to the eleven basic pattern forms belonging to F are numbered 1,2,...,11.

In addition, actual pattern names Form1,Form2,...,Form11 are used to specify basic pattern forms within F. Exp1,...,Exp11 are the corresponding pattern names of the Replace table, and denote macros which expand to retrieval terms used to replace occurrences of reducible expressions.

Init1,...,Initll are the respective pattern names within Init, and represent macros which expand to initialization code. The D table entries are divided into 11 separate groups, each of which is associated with a different basic form belonging to the F table. These groups are presented in the same order as the corresponding entries of F. Within each group associated with $Form_i$, $i = 1,...,11$, we list entries labeled by letters for every continuity parameter x of $Form_i$, and for each allowable modification to x. Each entry contains expressions for a parameter change pattern and a derivative code macro.

To avoid cluttering our tables with needless detail, the pattern entries displayed for these tables will not adhere strictly to the pattern language rules. Since a modified Subset1 parser can be made to recognize the literal symbols and tree structure of pattern expressions, literals will not be enclosed within quote marks, and brackets will not be used to specify tree structure. Thus, any occurrences of brackets will denote literal symbols. Parentheses, too, will denote literals instead of meta symbols.

For conciseness and readability, we will use macros to help generate pattern expressions. Macros are declared in the following way,

```
macro name(P1,P2,...,PN);
      <exp>
      <block>
endm
```

where name is the macro name, P1,...,PN are parameter names, exp is a pattern expression and block is a sequence of pattern name assignments. A macro use, name(text1,...,textN) causes $text_i$, i=1,...,N, to be substituted for each occurrence of P_i within exp and block. Note that a single token may be formed by concatenation from parts separated by the symbol @. A parameter may be part of such a token. Although pattern names defined outside of macros will be considered global within each table, all pattern names defined within a macro block are assumed to be local to each macro use. After macro expansion, exp will replace the macro invocation.

Pattern specifications used for the F table incorporated in the FD implementation discussed in Chapter 4 (C) normally contain a procedure name pattern of the form !pname immediately following each occurrence of a pattern variable pvar. Recall that when !pname is encountered during matching, the procedure pname which validates the sub-expression matched by pvar is executed.

For the sake of clarity, the F table shown below has been pruned of all procedure names. However, we use a naming convention for pattern variables which allows us to restore these missing procedure names systematically. We consider five such procedures in connection with an upgraded F table: Cvar(Indset), Dvar, Svar, Bvar, and Mvar.

The first three of these procedures have already been discussed on p. 426. Cvar is used for validating pattern variables which are continuity parameters. For each continuity parameter xi the procedure parameter Indset is the name of the induction set associated with xi. The following table gives the rule of correspondence between continuity parameters of the F table and the induction sets defined on p. 456.

Induction Set	Continuity Parameters
IV_1	x1, x2, x3, x4, x5
IV_3	x6, x7, x8, x9, x10, x11, x12, x13
IV_4	F1, F2, F3
IV_5	F8, F9
IV_6	x1, x2, x3, x4, x5
IV_7	F1, F2, F3

The routine Dvar will be used for validating discontinuity parameters. By convention all pattern variables of the F table whose name begins with 'q' are discontinuity parameters and should be followed by an occurrence of the procedure pattern !Dvar.

Svar is used to validate special parameters; these are pattern variables which begin with the letter 'k'.

In addition to the three procedures already discussed, we also require two new routines, Bvar and Mvar. Bvar validates all parameters which match variables bound to iterators.

All such parameters occurring within the F table use the name x. To implement Bvar we must ensure that whenever a term !Bvar is encountered just after an occurrence of a pattern variable x, the expression $\bar{x}$ matched by x represents a simple variable occurrence.

The routine Mvar is designed to validate pattern variables F4, F5, F6, and F7. Mvar must ensure that any subexpression matched by each of these parameters is a map variable.

Various other procedure names also prove useful in specifying patterns for the Init and D tables. Two of these procedures, zero and ℓt are used in connection with dotted pattern variables 'q'. We associated with q two special counters, #q and $q whose values are stored in Pfunc('#q') and Pfunc('$q'), where Pfunc is the pattern variable map used in either macro expansion of pattern matching. The counter #q associated with a particular pattern variable q. will have zero as its initial value, and will reflect the number of generated instances of q. . The counter $q will have the same value as #q. except when it is reset to some other value constrained by the condition $0 \leq$ Pfunc($q) $\leq$ Pfunc(#q).

We illustrate the use of these counters by considering the following pattern specifications,

(1)
```
Params1 ≡ !zero($y) Params2
Params2 ≡ ! ℓt($y, #q2) y.*, Params2 | y.*
```

in connection with macro expansion. Consider expansion of Params1 using a pattern variable map Pfunc in which Pfunc(#q2) has the value 3. Then !zero($y) will cause Pfunc($y) to be set to zero, and expansion to proceed with Params2. The procedure ℓt($y,#q2) will succeed whenever the value of $y is less than the value of #q2; otherwise it will fail.

The first time ℓt is called expansion succeeds, Pfunc(y1) is given a unique generated variable name n1, and the value of $y is incremented. After this the literal , is expanded, and Params2 is expanded recursively. Once again ℓt succeeds, and expansion of y.* results in assigning the value two to $y, and a new name n2 to Pfunc(y2). Next we expand the comma followed by Params2. This third attempt to call ℓt will also succeed; y.* will be expanded generating a third name n3; the comma will be expanded; and finally Params2 will be expanded. However, since $y has the value 3, ℓt will fail, and expansion will proceed with the right alternand, y.*. One last name n4 is generated before expansion terminates with the following result,

(2) n1, n2, n3, n4

representing seven sibling nodes of a tree. The sort of expansion just illustrated is useful in generating bound variables of forall loops within the derivative code sequences for Form1.

We use the procedure eq(x,y) to test the equality of the trees Pfunc(x) and Pfunc(y). eq succeeds whenever the tree comparison Equals(Pfunc(x), Pfunc(y)) (cf. Appendix E (ii)) holds and fails otherwise. For tree inequality we use the procedure ne(x,y).

Two other procedures used only in the D table are subst(k2,x,u) and strict. subst always succeeds, and substitutes all free occurrences of $\bar{x}$ within k2 with occurrences of $\bar{u}$. The routine strict is used within the D table in the following context, arg !strict, where $\overline{arg}$ can belong to either the induction sets IV1 or IV6. strict succeeds whenever $\overline{arg}$ belongs to IV6 and fails otherwise.

Elementary Form Table (F)

1. Form1 ≡ {x ∈ x1 '|' k}

k ≡ Conj & k | Conj

Conj ≡ F8.(x) = 0 |
F9.(x) ¬= 0 |
k5. |
k1. = q1. |
! x ∈ x2 |
! x ∉ x3 |
! k2 ∈ x4 |
! k3 ∉ x5 |
! x ∈ F1(Q2) |
! k4 ∈ F2(Q3) |
q4. ∈ F3.(x) |
! x < x6 |
! x ≤ x7 |
! x > x8 |
! x ≥ x9 |
! F4(x) < x10 |
! F5(x) ≤ x11 |
! F6(x) > x12 |
! F7(x) ≥ x13

Q2 ≡ q2., Q2 | q2.

Q3 ≡ q3., Q3 | q3.

macro Setform(op); {x ∈ x1 '|' F1(x,q) op F2(x)} *endm*

2. Form2 ≡ Setform(∈)

3. Form3 ≡ Setform(∉)

4. Form4 ≡ Setform(<)

5. Form5 ≡ Setform(≤)

6. Form6 ≡ Setform(>)

7. Form7 ≡ Setform(≥)

8. Form8 ≡ x1 + x2

9. Form9 ≡ [+: x ∈ x1]1

10. Form10 ≡ [+: x ∈ F1(Q2)]1

11. Form11 ≡ {x ∈ F10(Q5) '|' k}

    Q5 ≡ q5., Q5 | q5.

Replace Table

1. Exp1 ≡ E(Params) | E

   Params ≡ Param, Params | Param

   Param ≡ q1. | q2. | q3. | q4.

2. Exp2 ≡ E(q)

3. Exp3 ≡ E(q)

4. Exp4 ≡ E(q)

5. Exp5 ≡ E(q)

6. Exp6 ≡ E(q)

7. Exp7 ≡ E(q)

8. Exp8 ≡ E

9. Exp9 ≡ E

10. Exp10 ≡ E(Q2)

    Q2 ≡ q2., Q2 | q2.

11. Exp11 ≡ E(Mparams)

    Mparams ≡ Mparam, Mparams | Mparam
    Mparam ≡ q5. | q1. | q2. | q3. | q4.

```
Init Table
1.   Initl  ≡ Srts
                E* := nullset
                (∀x* ∈ x1, Iteradd '|' k)
                     Emap1 := Emap1 + {x} ort {x};
                end ∀;
     Emap1 ≡ !zero($k1,$w3) E(Params)
     Params ≡ Param, Params | Param
     Param  ≡ k1. | ! w1 | ! w2 | w3.
     Iteradd ≡ Iter, Iteradd | Iter
     Iter ≡ ! w1* ∈ {u* ∈ Project(#q2,F1) '|' x ∈ F1(u)} |
            ! w2* ∈ {u* ∈ Project(#q3,F2) '|' k4 ∈ F2(u)}|
              w3.* ∈ F3.(x)
     k ≡ Conj & k | Conj
     Conj ≡ ! x ∈ x2          |
            ! x ∉ x3          |
            ! k2 ∈ x4         |
            ! k3 ∉ x5         |
            ! x < x6          |
            ! x ≤ x7          |
            ! x > x8          |
            ! x ≥ x9          |
            ! F4(x) < x10     |
            ! F5(x) ≤ x11     |
            ! F6(x) > x12     |
            ! F7(x) ≥ x13     |
              F8.(x) = 0      |
```

```
            F9.(x) ¬= 0    |
            k5.
      Srts ≡ Srt Srts | Srt
      macro Sort(set,op,id);
         !  Sortas(set, pred@id@u*,succ@id@u);
            xmin@id@u := [min: w* ∈ arg '|' w ≥ x@id]w;
            pred@id@u(Ω) := [max: w ∈ arg]w;
      endm;
      macro Sortr(op,id); Sort(xl,op,id) endm
      macro Sortf(op,id,num); Sort(F@num(xl),op,id) endm
      Srt ≡ Sortr(≥,6)        |
            Sortr(>,7)        |
            Sortr(>,8)        |
            Sortr(≥,9)        |
            Sortf(≥,10,4)     |
            Sortf(>,11,5)     |
            Sortf(>,12,6)     |
            Sortf(≥,13,7)
2.    Init2 ≡ E* := nullset;
            (∀x* ∈ Dom F1, y* ∈ Dom F1{x} '|'
                              x ∈  x1 & F1(x,y) ∈ F2(x))
                   E(y) := E(y) + {x} ort {x};
             end ∀;
3.    Init3 ≡ E* := nullset;
            (∀x* ∈ Dom F1, y* ∈ Dom F1{x} '|'
                              x ∈ x1 & F1(x,y) ∉ F2(x))
                E(y) := E(y) + {x} ort {x};
             end ∀;
```

```
macro Initform(op1,op2);
      E* := nullset;
      (∀x* ∈ Dom F1, y* ∈ Dom F1{x} '|' F1(x,y) op1 F2(x))
           E(y) := E(y) + {x} ort {x};
      end ∀;
      xmin@u*   := nullset;
      (∀x ∈ x1)
           Sortas(Dom F1{x}, pred@u, succ@u, x);
           pred@u(x,Ω) := [max: y ∈ Dom F1{x}]y;
           xmin@u(x) := [min: y ∈ Dom F1{x} '|' y op2 F2(x)]y;
      end ∀;
endm
4.   Init4  ≡ Initform(<,≥)
5.   Init5  ≡ Initform(≤,>)
6.   Init6  ≡ Initform(>,>)
7.   Init7  ≡ Initform(≥,≥)
8.   Init8  ≡ E* := x1 + x2;
9.   Init9  ≡ E* := [+: x* ∈ x1]1;
10.  Straightforward Initialization
  a. Init10 ≡ E* := nullset;
               (∀[Params1] ∈ Project(#q2,F1))
                  E(Params3) := [+: x* ∈ F1(Params3)]1;
               end ∀;
     Params1  ≡ !zero($y) Params2
     Params2  ≡ !Lt($y,#q2) y.*, Params2 | y.*
     Params3  ≡ !zero($y) Params4
     Params4  ≡ y., Params4 | y.
```

Differential Initialization

b. Parameter change:

Fl := *nullset*;

Prederivative:

E* := *nullset*;

c. Parameter change:

Fl(Params5) := Fl(Params5) + Δ *ort* Δ;

Params5 ≡ !zero($y) Params6

Params6 ≡ !Lt($y,#q2) y., Params6 | y.

Prederivative:

E(Params5) := E(Params5) + 1 *ort* 1;

11. Initll ≡ E* := *nullset*;

(∀[Params7] ∈ Project(#q1+#q2+#q3+#q4+#q5,Fl))

E(Params3) := {x* ∈ Fl(Params3) '|' k};

end ∀;

Params7 ≡!zero($y) Params8

Params8 ≡ !Lt($y,#q1+#q2+#q3+#q4+#q5)

y.*, Params8 | y.*

Derivative Table (D)

```
macro Itin(arg);
     Δ - arg !strict | Δ
endm
macro Itout(arg);
     Δ * arg !strict | Δ
endm
macro Iteradd(skip,num);
   Iterad
   Iterad ≡ Itera,  Iterad | Itera
   Itera  ≡ !ne(skip,F1) ! [P6] ∈
                  {[P5] ∈ Project(#q2,F1) '|' x ∈ F1(u)}      |
            !ne(skip,F2) ! [P7] ∈
                  {[P5] ∈ Project(#q3,F2) '|' k4 ∈ F2(u)}     |
            !ℓt($F3,num-1) w3.* ∈ F3.(x)                      |
            !eq(skip,F3) !eq($F3,num-1) w3.* ∈ Itin(F3.(x)) |
            !ℓt($F3,#F3) w3.* ∈ F3.(x)
endm
macro Itersub(skip,num);
   Itersu
   Itersu ≡ Iters, Itersu | Iters
   Iters  ≡ !ne(skip,F1) ! [P6] ∈
                  {[P5] ∈ Project(#q2,F1) '|' x ∈ F1(u)} * Dom E |
            !ne(skip,F2)! [P7]
                  {[P5] ∈ Project(#q3,F2) '|' k ∈F2(u)}* Dom E{w1}|
```

```
            !lt(#F3,num-1) w3.* ∈ F3.(x) * Dom E{Params1}  |
            !eq(skip,F3) !eq($F3,num-1) w3.*∈Itout(F3.(x))*
                                Dom E{Params1}              |
            !lt($F3,#F3) w3. ∈ F3.(x) * Dom E{Params1}
endm
Params1 ≡ !zero($w3,$w1,$w2) Params2
Params2 ≡ Param2, Params2 | Param2
Param2  ≡ w1.|
          w2.|
          w3.
Params ≡ !zero($K1,$w3,$w1,$w2) Parameters
Parameters   Param, Parameters | Param
Param ≡ k1.|  w1.|   w2.| w3.
P5 ≡ !zero($u) P8
P8 ≡ u.*, P8 | u.*
P6 ≡ !zero($w1) P9
P9 ≡ w1.*, P9 | w1.*
P7 ≡ !zero($w2) P10
P10 ≡ w2.*, P10 | w2.*
macro k(skip,num);
   bool
   bool ≡ Conj & bool | Conj
   Conj ≡ !ne(skip,x2) ! x ∈ x2  |
          !ne(skip,x3) ! x ∉ x3  |
          !ne(skip,x4) ! k2 ∈ x4 |
          !ne(skip,x5) ! k3 ∈ x5 |
```

```
!ne(skip,x6) ! x  < x6                                    |
!ne(skip,x7) ! x  ≤ x7                                    |
!ne(skip,x8) ! x  > x8                                    |
!ne(skip,x9) ! x  ≥ x9                                    |
!ne(skip,x10) ! F4(x) > x10                               |
!ne(skip,x11) ! F5(x) ≤ x11                               |
!ne(skip,x12) ! F6(x) > x12                               |
!ne(skip,x13) ! F7(x) ≥ x13                               |
!lt($F8,num-1) F8.(x) = 0                                 |
!eq(skip,F8) !eq($F8,num-1) !Inc($F8)                     |
!lt($F8,#F8) F8.(x) = 0                                   |
!lt($F9,num-1) F9.(x) ¬= 0                                |
!eq(skip,F9) !eq($F9,num-1) !Inc($F9)                     |
!lt($F9,#F9) F9.(x) ¬= 0                                  |
k5.
```

endm

1a. Parameter change:

xl := xl + Δ;

Prederivative:

(∀x* ∈ Itin(xl), Iteradd '|' k)

E(Params) := E(Params) + {x};

end ∀;

b. Parameter change:

xl := xl - Δ;

Prederivative:

(∀x* ∈ Itout(xl), Itersub '|' k)

E(Params := E(Params) - {x};

end ∀;

c. Parameter change:

x2 := x2 + Δ;

Prederivative:

(∀x* ∈ Itin(x2), Iteradd '|' k(x2) & x ∈ xl)

E(Params) := E(Params) + {x};

end ∀;

d. Parameter change:

x2 := x2 - Δ;

Prederivative:

(∀x* ∈ Itout(x2), Itersub '|' k(x2) & x ∈ xl)

E(Params) := E(Params) - {x};

end ∀;

e. Parameter change:

x3 := x3 + Δ;

Prederivative:

(∀x*∈Itin(x3), Itersub '|' k(x3) & x ∈ x1)

E(Params) := E(Params) - {x};

end ∀;

f. Parameter change:

x3 := x3 - Δ;

Prederivative:

(∀x*∈Itout(x3), Iteradd '|' k(x3) & x ∈ x1)

E(Params) := E(Params) + {x};

end ∀;

g. Parameter change:

x4 :=x4 -Δ;

Prederivative:

(∀y* ∈ Itin(x4), x* ∈ {u* ∈ x1 '|' !subst(k2,x,u) = y},

Iteradd '|' k(x4))

E(Params) := E(Params) + {x};

end ∀;

h. Parameter change:

x4 := x4 - Δ;

Prederivative:

(∀y* ∈ Itout(x4), x* ∈ {u* ∈ x1 '|' !subst(k2,x,u) = y},

Itersub '|' k(x4))

E(Params) := E(Params) - {x};

end ∀;

```
i.  Parameter change:
    x5 := x5 + Δ;
    Prederivative:
    (∀y* ∈ Itin(x5), x* ∈ {u* ∈ xl '|' !subst(k3,x,u) = y},
        Itersub '|' k(x5))
        E(Params) := E(Params) - {x};
    end ∀;
j.  Parameter change:
    x5 := x5 - Δ;
    Prederivative:
    (∀y* ∈ Itout(x5), x* ∈ {u* ∈ xl '|' !subst(k3,x,u) = y},
       Iteradd '|' k(x5))
       E(Params) := E(Params) + {x};
    end ∀;
k.  Parameter change:
    F3@$F3(x) := F3@$F3(x) + Δ;
    Prederivative:
    if x ∈ xl then
       (∀Iteradd(F3,$F3) '|' k)
          E(Params) := E(Params) + {x};
       end ∀;
    endif;
l.  Parameter change:
    F3@$F3(x) := F3@$F3(x) - Δ;
```

```
   Prederivative:
   if x ∈ xl then
      (∀Itersub(F3,$F3) '|' k)
         E(Params) := E(Params) - {x};
      end ∀;
   endif;
m. Parameter change:
   Fl(Paramsl0) := Fl(Paramsl0) + Δ;
   Paramsl0 ≡ !zero($wl) Paramsll
   Paramsll ≡ !Lt($wl,#q2) wl. Paramsll | wl.
   Prederivative:
   (∀x* ∈ Itin(Fl(Paramsl0)), Iteradd(Fl) '|' x ∈ xl & k)
      E(Params) := E(Params) + {x};
   end ∀;
n. Parameter change:
   Fl(Paramsl0) := Fl(Paramsl0) - Δ;
   Prederivative:
   (∀x* ∈ Itout(Fl(Paramsl0)), Itersub(Fl) '|' x ∈ xl & k)
      E(Params) := E(Params) - {x};
   end ∀;
o. Parameter change:
   F2(Paramsl2) := F2(Paramsl2) + Δ;
   Paramsl2 ≡ !zero($w2) Paramsl3
   Paramsl3 ≡ !Lt($w2,#q3) w2. Paramsl3 | w2.
   Prederivative:
   (∀y* ∈ Itin(F2(Paramsl2)), x* ∈
                        {u* ∈ xl '|' !subst(k4,x,u) = y},
      Iteradd(F2) '|' k)
      E(Params) := E(Params) + {x};
   end ∀;
```

```
 p.  Parameter change:
     F2(Paramsl2) := F2(Paramsl2) - Δ;
     Prederivative:
     (∀y* ∈ Itout(F2(Paramsl2)), x* ∈
                          {u* ∈ xl '|' subst(k4,x,u) = y},
        Itersub(F2) '|' k)
        E(Params) := E(Params) - {x};
     end ∀;
macro Dsuccr(relop,setop,id);
   (while xmin@id@u relop x@id + Δ)
      (∀x* := xmin@id@u,
         Iterator '|' k(x@id))
         E(Params) := E(Params) setop {x};
      end ∀;
      xmin@id@u := succ@id@u(xmin@id@u);
   endwhile;
   Iterator ≡ !eq(setop,+) Iteradd | Itersub
endm
macro Dpredr(relop,setop,id);
   (while pred@id@u(xmin@id@u) relop x@id - Δ)
      xmin@id@u := pred@id@u(xmin@id@u);
      (∀x* := xmin@id@u,
         Iterator '|' k(x@id))
         E(Params) := E(Params) setop {x};
      end ∀;
   endwhile;
   Iterator ≡ !eq(setop,+) Iteradd | Itersub
endm
```

```
macro Dsuccf(relop,setop,id,arg);
   (while xmin@id@u relop x@id + Δ)
      (∀x* ∈ {u* ∈ xl '|' arg(u) = xmin@id@u},
         Iterator '|' k(x@id))
         E(Params) := E(Params) setop {x};
      end ∀;
      xmin@id@u := succ@id@u(xmin@id@u);
   endwhile;
   Iterator ≡ !eq(setop,+) Iteradd | Itersub
endm
macro Dpredf(relop,setop,id,arg);
   (while pred@id@u(xmin@id@u) relop x@id - Δ)
      xmin@id@u := pred@id@u(xmin@id@u);
      (∀x* ∈ {u* ∈ xl '|' arg(u) = xmin@id@u},
         Iterator '|' k(x@id))
         E(Params) := E(Params) setop {x};
      end ∀;
   endwhile;
   Iterator ≡ !eq(setop,+) Iteradd | Itersub
endm
```

q. Parameter change:

```
   x6 := x6 + Λ;
   Prederivative:
   Dsuccr(<,+,6)
```

r. Parameter change:

x6 := x6 - Δ;

Prederivative:

Dpredr(≥,-,6)

s. Parameter change:

x7 := x7 + Δ;

Prederivative:

Dsuccr(≤,+,7)

t. Parameter change:

x7 := x7 - Δ;

Prederivative:

Dpredr(>,-,7)

u. Parameter change:

x8 := x8 + Δ;

Prederivative:

Dsuccr(≤,-,8)

v. Parameter change:

x8 := x8 - Δ;

Prederivative:

Dpredr(>,+,8)

w. Parameter change:

x9 := x9 + Δ;

Prederivative:

Dsuccr(<,-,9)

x. Parameter change:

x9 := x9 - Δ;

Prederivative:

Dpredr(≥,+,9)

y. Parameter change:

x10 := x10 + Δ;

Prederivative:

Dsuccf(<,+,10,F4)

z. Parameter change:

x10 := x10 - Δ;

Prederivative:

Dpredf(≥,-,10,F4)

aa. Parameter change:

x11 := x11 + Δ;

Prederivative:

Dsuccf(≤,+,11,F5)

bb. Parameter change:

x11 := x11 - Δ;

Prederivative:

Dpredf(>,-,11,F5)

cc. Parameter change:

x12 := x12 + Δ;

Prederivative:

Dsuccf(≤,-,12,F6)

dd. Parameter change:

x12 := x12 - Δ;

Prederivative:

Dpredf(>,+,12,F6)

ee. Parameter change:

x13 := x13 + Δ;

Prederivative:

Dsuccf(<,+,13,F7)

ff. Parameter change:

```
x13 - Δ;
Prederivative:
Dpredf(≥,-,13,F7)
```

gg. Parameter change:

```
F8@$F8(x) := F8@$F8(x) + Δ;
Prederivative:
if x ∈ x1 & F8@$F8(x) = 0 then
   (∀Itersub '|' k(F8,$F8))
      E(Params) := E(Params) - {x};
   end ∀;
endif;
```

hh. Parameter change:

```
F8@$F8(x) := F8@$F8(x) - Δ;
Prederivative:
if x ∈ x1 & F8@$F8(x) = Δ then
   (∀Iteradd '|' k(F8,$F8))
      E(Params) := E(Params) + {x};
   end ∀;
endif;
```

ii. Parameter change:

```
F9@$F9(x) := F9@$F9(x) + Δ;
Prederivative:
if x ∈ x1 & F9@$F9(x) = 0 then
   (∀Iteradd '|' k(F9,$F9))
      E(Params) := E(Params + {x};
   end ∀;
endif;
```

jj. Parameter change:

```
F9@$F9(x) := F9@$F9(x) - Δ;
Prederivative:
if x ∈ xl & F9@$F9(x) = Δ then
   (∀Itersub '|' k(F9,$F9))
      E(Params) := E(Params) - {x};
   end ∀;
endif;
```

2a. Parameter change:

F2(y) :=F2(y) + Δ;

Prederivative:

if y ∈ xl *then*

(∀x* ∈ Itin(F2(y)),u* ∈ {w* ∈ *Dom* Fl{y} '|' Fl(y,w) = x})

E(u) := E(u) + {x};

end ∀;

endif;

b. Parameter change:

F2(y) := F2(y) - Δ;

Prederivative:

if y ∈ xl *then*

(∀x* ∈ Itout(F2(y)), u* ∈ {w* ∈ *Dom* Fl{y} '|' Fl(y,w)=x})

E(u) := E(u) - {x};

end ∀;

endif;

3a. Parameter change:

F2(y) := F2(y) + Δ;

Prederivative:

if y ∈ xl *then*

(∀x* ∈ Itin(F2(y)), u* ∈ {w* ∈ *Dom* Fl{y} '|' Fl(y,w)=x})

E(u) := E(u) - {x};

end ∀;

endif;

```
b.  Parameter change:
    F2(y) := F2(y) - Δ;
    Prederivative:
    if y ∈ xl then
      (∀x* ∈ Itout(F2(y)), u* ∈ {w* ∈ Dom Fl{y} '|' Fl(y,w)=x})
        E(u) := E(u) + {x};
      end ∀;
    endif;

macro xsucc(relop,setop);
   (while xmin@u(y) relop F2(y) + Δ)
      (∀x* ∈ {w* ∈ Dom Fl{y} '|' Fl(y,w) = xmin@u(y)})
        E(x) := E(x) setop {y};
      end ∀;
      xmin@u(y) := succ@u(y,xmin@u(y));
   endwhile;
endm

macro xpred(relop,setop);
   (while pred@u(y,xmin@u(y)) ≥ F2(y) - Δ)
      xmin@u(y) := pred@u(y,xmin@u(y));
      (∀x* ∈ {w* ∈ Dom Fl{y} '|' Fl(y,w) = xmin@u(y)})
        E(x) := E(x) setop {y};
      end ∀;
   endwhile;
endm
```

4a. Parameter change:

F2(y) := F2(y) + Δ;

Prederivative:

xsucc(<,+)

b. Parameter change:

F2(y) := F2(y) - Δ;

Prederivative:

xpred(≥,-)

5a. Parameter change:

F2(y) := F2(y) + Δ;

Prederivative:

xsucc(≤,+)

b. Parameter change:

F2(y) := F2(y) - Δ;

Prederivative:

xpred(>,-)

6a. Parameter change:

f2(y) := F2(y) + Δ;

Prederivative:

xsucc(≤,-)

b. Parameter change:

F2(y) := F2(y) - Δ;

Prederivative:

xpred(>,+)

7a. Parameter change:

F2(y) := F2(y) + Δ;

Prederivative:

xsucc(<,-)

b. Parameter change:

F2(y) := F2(y) + Δ;

Prederivative:

xpred(≥,+)

8a. Parameter change:

x1 := x1 + Δ;

Prederivative:

E := E + Δ;

b. Parameter change:

x1 := x1 - Δ;

Prederivative:

E := E - (Δ - x2);

c. Parameter change:

x2 := x2 + Δ;

Prederivative:

E := E + Δ;

d. Parameter change:

x2 := x2 - Δ;

Prederivative:

E := E - (Δ - x1);

9a. Parameter change:

x1 := x1 + Δ;

Prederivative:

E := E + [+: x ∈ itin(x1)]1;

b. Parameter change:

x1 := x1 - Δ;

Prederivative:

E := E - [+: x ∈ itout(x1)]1;

10a. Parameter change:

```
Fl(Params3) := Fl(Params3) + Δ;

Params3 ≡ !zero($y) Params4

Params4  ≡ !Lt($y,#q2) y., Params4 | y.
```

Prederivative:

```
E(Params3) := E(Params3) + [+: x* ∈ Init(Fl(Params3))]1;
```

b. Parameter change:

```
Fl(Params3) := Fl(Params3) - Δ;
```

Prederivative:

```
E(Params3) := E(Params3)- [+: x* ∈ Inout(Fl(Params3))]1;
```

APPENDIX D

VARIOUS ELEMENTARY AND COMPOUND SET THEORETIC TRANSFORMATIONS

In this section we give a list of auxiliary set theoretic transformations likely to be useful supplements to formal differentiation around which our proposed implementation will be built. This list includes transformations likely to aid in performing preparatory and cleanup tasks arising before and after formal differentiation. Although most of these rules lie at a relatively low level, at the end of the present appendix we describe a way to collect simple rules into 'rule groups' applicable automatically and collectively over a region.

Each transformation we consider will be written as a rewrite rule in one of two forms. We write LHS ⇒ RHS to indicate that the LHS pattern can be replaced by RHS; the second form LHS ⇔ RHS designates a production allowing for replacement of either RHS by LHS or of LHS by RHS. The notation <k,x \y> indicates that all occurrences of the term x in k are to be replaced by y.

Unless otherwise specified we assume that all transformable expressions are applicative, and consequently side effect free. For this reason, in theory the usual transformations (e.g., commutative laws) which can rearrange the order of expression of computations can be

expected to hold. Since in SETL, primitive operations consider boolean arguments to be *false* if not explicitly *true*, we expect that standard identities such as

$$\exists x \in s \mid f(x) > g(x) \Leftrightarrow \neg \forall x \in s \mid \neg (f(x) > g(x))$$

are preserved. Due to finite computer data representation and machine dependent error condition handling, issues of code motion safety frequently pose obstacles to nontrivial arithmetic transformations (e.g. distributive laws may not hold), but this does not concern us since our primary interest is in expressions involving finite sets and set theoretic relations. However, even when arithmetic operations appear, the order of evaluation may sometimes be changed without sacrificing accuracy. (cf. W1 for a further discussion of safety problems in formal differentiation).

Some of the transformations listed here will rearrange the execution order of code C only when the Usetodef and Deftouse maps do not change, a precondition which we call 'transformational disjointness'.

i. Simple Set Identities (all arguments are set valued)

	COMMUTATIVE Laws		ASSOCIATIVE Laws
C1.	$S * T \Leftrightarrow T * S$	A1.	$S * (T * Q) \Leftrightarrow (S * T) * Q$
C2.	$S + T \Leftrightarrow T + S$	A2.	$S + (T + Q) \Leftrightarrow (S + T) + Q$

Idempotent

I1. S * S ⇔ S

A2. S + S ⇔ S

Neutral Element

N1. S + *nullset* ⇔ S

N2. S - *nullset* ⇔ S

Rules for *Nullset*

N3. S * *nullset* ⇔ *nullset*

N4. S - S ⇔ *nullset*

DISTRIBUTIVE

D1. S * (T + Q) ⇔ (S * T) + (S*Q)

D2. S + (T * Q) ⇔ (S + T) * (S+Q)

D3. S * (T - Q) ⇔ (S * T) - (S*Q)

D4. (S + T) - Q ⇔ (S - Q) + (T-Q)

MISCELLANEOUS Rules

M1. s - (t + Q) ⇔ (s - t) - Q ⇔ (s - Q) - t

M2. s + (t - Q) ⇔ (s + t) - (Q - s)

M3. s - (t - Q) ⇔ (s - t) + (s * Q)

M4. s - (t * Q) ⇔ (s - t) + (s - Q)

M5. s * (t - Q) ⇔ (s - Q) * t

M6. s - t ⇔ s - (t * s) ⇔ (s - t) - t

M7. s ⇔ (s + t) - (t - s) ⇔ (s - t) + (s * t)

M8. s + t ⇔ s + (t - s)

M9. s * t ⇔ s - (s - t)

M10. (S - T) * T ⇔ *nullset*

Rules which Require T *INCS* S as an Enabling Condition

E1. S - T ⇔ *nullset*

E2. S * T ⇔ S

E3. S + T ⇔ T

E4. S ¬= T ⇔ T - S ¬= *nullset*

Tautologies

T1. T *INCS nullset*

T2. T *INCS* T * S

T3. T *INCS* T - S

T4. T + S *INCS* S

ii. Boolean and Relational Identities

	COMMUTATIVE		ASSOCIATIVE
C1.	K & J ⇔ J & K	A1.	K & (J & L) ⇔ (K & J) & L
C2.	K *or* J ⇔ J *or* K	A2.	K *or* (J *or* L) ⇔ (K *or* J) *or* L

	Idempotent		Neutral Element
I1.	K & K ⇔ K	N1.	K & *true* ⇔ K
I2.	K *or* K ⇔ K	N2.	K *or* *false* ⇔ K

	Rules for TRUE, False		DISTRIBUTIVE
N3.	K & *False* ⇔ *False*	D1.	K & (J *or* L) ⇔ (K & J) *or* (K & L)
N4.	K *or* *True* ⇔ *True*	D2.	K *or* (J & L) ⇔ (K *or* J) & (K *or* L)

NEGATION

N5. ¬ ¬ K ⇔ K

N6. ¬ *False* ⇔ *True*

DE MORGAN'S LAWS

M1. K *or* J ⇔ ¬ (¬ K & ¬ J)

M2. K & J ⇔ ¬(¬ K *or* ¬ J)

	RELATIONAL and Negation		Relational
R1.	¬(K ¬= J) ⇔ (K = J)	R5.	I < M ⇔ M > I
R2.	¬(K ∉ S) ⇔ K ∈ S	R6.	I ≤ M ⇔ M ≥ I
R3.	¬(I > M) ⇔ I ≤ M } ≠	R7.	I = M ⇔ M = I
R4.	¬(I ≥ M) ⇔ I < M } ≠	R8.	I ¬= M ⇔ M ¬= I

≠ I and M must be integers

MORE COMPLICATED RULES

$K = J \Leftrightarrow K \in \{J\}$

$[or\colon x_1 \in s_1, \ldots, x_n \in s_n | K_1]\ [or\colon y_1 \in t_1, \ldots y_m \in t_m | k_2]k_3$

$\Updownarrow$

$[or\colon x_1 \in s_1, \ldots, x_n \in s_n, y_1 \in t_1, \ldots, y_m \in t_m | k_1 \ \& \ k_2]k_3$

$[\&\colon x_1 \in s_1, \ldots, x_n \in s_n | k_1][\&\colon y_1 \in t_1, \ldots, y_m \in t_m | k_2]k_3$

$\Updownarrow$

$[\&\colon x_1 \in s_1, \ldots, x_n \in s_n, y_1 \in t_1, \ldots, y_m \in t_m | k_1 \ \& \ k_2]k_3$

DE MORGAN'S LAWS

$[\&\colon x_1 \in s_1, \ldots, x_n \in s_n | k_1]k_2$

$\Updownarrow$

$\neg[or\colon x_1 \in s_1, \ldots, x_n \in s_n | k_1]\neg k_2$

$[or\colon x_1 \in s_1, \ldots, x_n \in s_n | k_1]k_2$

$\Updownarrow$

$\neg[\&\colon x_1 \in s_1, \ldots, x_n \in s_n | k_1]\neg k_2$

DISTRIBUTIVE LAWS

$[or\colon x_1 \in s_1, \ldots, x_n \in s_n | k_1](k_2 \overset{\&}{or} k_3)$

$\Updownarrow$

$[or\colon x_1 \in s_1, \ldots, x_n \in s_n | k_1]k_2 \overset{\&}{or} [or\colon x_1 \in s_1, \ldots x_n \in s_n | k_1]k_3$

$[\&\colon x_1 \in s_1, \ldots, x_n \in s_n | k_1](k_2 \overset{\&}{or} k_3)$

$\Updownarrow$

$[\&\colon x_1 \in s_1, \ldots, x_n \in s_n | k_1]k_2 \overset{\&}{or} [\&\colon x_1 \in s_1, \ldots, x_n \in s_n | k_1]k_3$

SIMPLIFICATION

$[^{\&}_{or}: x_1 \in s_1 , \ldots, x_n \in s_n | k_1] k_2 \Leftrightarrow k_2$

if $x_1, \ldots, x_n$ are not free in k_2 and

$\{<x_1, \ldots, x_n>, x_1 \in s_1 , \ldots, x_n \in s_n | k_1\} \neg = nullset$

$[^{or}_{\&}: x_1 \in s_1 , \ldots, x_n \in s_n | k_1] k_2$

$\Updownarrow$

$[^{or}_{\&}: x_1 \in s_1 , \ldots, x_n \in s_n \mid k_1 \ \& \ k_2]$ *True*

iii. Set Former Manipulation and Simplification Rules

DISTRIBUTIVE LAWS

$\{e: x_1 \in s_1 , \ldots, x_n \in s_n | k_1 \text{ or } k_2\}$

$\Updownarrow$

$\{e: x_1 \in s_1 , \ldots, x_n \in s_n | k_1\} + \{e: x_1 \in s_1 , \ldots, x_n \in s_n | k_2\}$

$\{e: x_1 \in s_1 , \ldots, x_n \in s_n | k_1 \ \& \ k_2\}$

$\Updownarrow$

$\{e: x_1 \in s_1 , \ldots, x_n \in s_n | k_1\} * \{e: x_1 \in s_1 , \ldots, x_n \in s_n | k_2\}$

$\{e: x_1 \in s_1 , \ldots, x_n \in s_n | [or: y_1 \in t_1 , \ldots, y_m \in t_m] k\}$

$\Updownarrow$ transformational disjointness required

$[+: y_1 \in t_1 , \ldots, y_m \in t_m] \{e_1: x_1 \in s_1 , \ldots, x_n \in s_n | k\}$

$\{e: x_1 \in s_1 , \ldots, x_n \in s_n | [\&: y_1 \in t_1, \ldots, y_m \in t_m] K\}$

$\Updownarrow$ transformational disjointness required

$[*: y_1 \in t_1 , \ldots, y_m \in t_m] \{e: x_1 \in s_1 , \ldots, x_n \in s_n\}$

$[+: x_1 \in s_1, \ldots, x_n \in s_n | K]\{e\}$

$\Downarrow$

$\{e: x_1 \in s_1, \ldots, x_n \in s_n | K\}$

$A := exp(\{e: x_1 \in s_1, \ldots, x_n \in s_n | K\})$

$\Updownarrow$ A must not be free in RHS of assignment;

A := *nullset*; Transformational disjointness is

$(\forall x_1 \in s_1, \ldots, x_n \in s_n | K)$ required; *exp* is an expression

A := A + *exp*({e}); in which

END ∀; $exp(\{e_1, e_2\}) = exp(\{e_1\}) + exp(\{e_2\})$ holds.

$[or: x_1 \in s_1, \ldots, x_n \in s_n | K](J \overset{\in}{\notin} T)$

$\Updownarrow$

$J \overset{\in}{\notin} [+: x_1 \in s_1, \ldots, x_n \in s_n | K]T$

where $x_1, \ldots, x_n$ are not free in J.

Simplification Rules (basic to all iterative operations)

s1. $\{e: x_1 \in s_1, \ldots, x_J \in \{p\}, \ldots, x_n \in s_n | k\}$

$\Downarrow$

$\{<e, x_J \backslash p>: x_1 \in s_1, \ldots, x_{J-1} \in s_{J-1}, x_{J+1} \in <s_{J+1}, x_J \backslash p>, \ldots,$

$x_n \in <s_n, x_J \backslash p> \mid <k, x_J \backslash p>\}$

s2. $\{e: x_1 \in s_1, \ldots, x_J \in s_J, \ldots, x_n \in s_n | x_J \in Q \ \& \ K\}$

$\Updownarrow$ when s_J *INCS* Q e.g., Q = {p}

$\{e: x_1 \in s_1, \ldots, x_J \in Q, \ldots, x_n \in s_n | K\}$

where Q and s_J do not have free occurrences of $x_{J+1}, \ldots, x_n$.

s3. $\{e: x_1 \in s_1, \ldots, x_J \in s_J, \ldots, x_I \in s_I, \ldots, x_n \in s_n | k\}$

$\{e: x_1 \in s_1, \ldots, x_I \in s_J, \ldots, x_J \in s_J, \ldots, x_n \in s_n | k\}$

where s_I has no free occurrences of $x_J, \ldots, x_{I-1}$,
s_J has no free occurrences of $x_{J+1}, \ldots, x_I$,
and for $L = J+1, \ldots, I-1$, s_L does not have free occurrences of x_J or x_I.

s4. $\{e: x_1 \in s_1, \ldots, x_J \in$ *nullset* $, \ldots, x_n \in s_n | k\}$

$\Downarrow$

nullset

iv. Forall LOOP LAWS

$(\forall x_1 \in s_1, \ldots, x_n \in s_n | k)$ *BLOCK END* $\forall$;

$\Updownarrow$

$(\forall x_1 \in s_1)$
$(\forall x_2 \in s_2, \ldots, x_n \in s_n | k)$ *BLOCK END* $\forall$;
END $\forall$;

$(\forall x \in s)$ *BLOCK1 END* $\forall$;
$(\forall x \in s)$ *BLOCK2 END* $\forall$;

$\Downarrow$

$(\forall x \in s)$ *BLOCK1 BLOCK2 END* $\forall$;

provided *BLOCK1*(x) commutes with *BLOCK2*(y)
for every $x, y \in s$ & $x \neq y$;

Obvious analogues of the three set former simplification rules discussed previously in iii can be made for 'forall'

loops and can result in diminishing the size of or removing iterators.

DISTRIBUTIVE TRANSFORMATIONS

$(\forall x_1 \in s_1, \ldots, x_n \in s_n | k)$ if P then *BLOCK* *ENDIF*; *END* $\forall$;

$\Downarrow$

$(\forall x_1 \in s_1, \ldots, x_n \in s_n | K \ \& \ P)$ *BLOCK* *END* $\forall$;

$(\forall x_1 \in s_1, \ldots, x_n \in s_n | P \ \& \ K)$ *BLOCK* *END* $\forall$;

$\Downarrow$

if P then $(\forall x_1 \in s_1, \ldots, x_n \in s_n | K)$ *BLOCK* *END* $\forall$;
ENDIF;

where $x_1, \ldots, x_n$ do not occur free in P.

v. Commonly Occurring Transformations Preparatory to Formal Differentiation

P1. $\#\{x \in s | K\} \Rightarrow [+: x \in s | k] 1$

P2. $\exists x \in s | k \Rightarrow ([+: x \in s | k] 1) = 0$

P3. $\forall x \in s | k \Rightarrow ([+: x \in s | \neg k] 1) = 0$

} where x is not used beyond the quantifier

P4. $\exists x_1 \in s_1, \ldots, x_n \in s_n | k \Leftrightarrow [or: x_1 \in s_1, \ldots, x_n \in s_n] k$

P5. $\forall x_1 \in s_1, \ldots, x_n \in s_n | k \Leftrightarrow [\&: x_1 \in s_1, \ldots, x_n \in s_n] k$

} where

$x_1, \ldots, x_n$ are not used beyond quantifier

P6. $\exists x \in s | k \Rightarrow \exists x \in \{y \in s | \langle k, x \backslash y \rangle\}$

P7. $\forall x \in s | k \Rightarrow \forall x \in \{y \in s | \langle k, x \backslash y \rangle\}$

P8. $s \neq_{=} nullset \Rightarrow ([+: y \in s]1) \neq_{=} 0$

P9. $s\ INCS\ R \Rightarrow ([+: y \in s | y \notin R]1) = 0$

(P6–P9: y is a new variable name which is generated.)

P10. $S = R \Rightarrow (S\ INCS\ R)\ \&\ (R\ INCS\ S)$

P11. $\{x \in S | K\} \Leftrightarrow S - \{x \in S \mid \neg K\}$

P12. $\{x \in s | K_1\ \&\ K_2\} \Leftrightarrow \{x \in S | K_1\} - \{x \in S \mid \neg K_2\}$

P13. $[op: x \in S | K_1]e \Rightarrow [op: x \in \{y \in S \mid \langle K_1, x \backslash y \rangle\}]e$

P14. $[or: x \in S]K \Rightarrow ([+: x \in S | K]1) \neg= 0$

P15. $[\&: x \in S]K \Rightarrow ([+: x \in S \mid \neg K]1) = 0$

P16. $S \neg= nullset \Rightarrow ([+: y \in S]1) \neg= 0$

P17. $S * T \Rightarrow \{x \in S | x \in T\}$

P18. $S - T \Rightarrow \{x \in S | x \notin T\}$

P19. $S := S + \Delta \Rightarrow (\forall x \in (\Delta - S))$

$S := S + \{x\};$

end ∀;

P20. $S := S - \Delta \Rightarrow (\forall x \in (\Delta * S))$

$s := s - \{x\};$

end ∀;

vi. Productions Derivable from Previous More Basic Rules and Useful for Cleanup After Formal Differentiation

C1. $\{z \in \{y\} | K\} \Rightarrow$ if $\langle K, z \backslash y \rangle$ then $\{y\}$ else *nullset*

C2. $(\forall z \in \{y\} | K)$ *BLOCK END* ∀; ⇒ if $\langle K, z \backslash y \rangle$ then $\langle$*BLOCK*, $z \backslash y \rangle$ *ENDIF*;

C3. $\exists z \in \{y\} | K \Rightarrow \langle K, z \backslash y \rangle$

C4. $\forall z \in \{y\} | K \Rightarrow \langle K, z \backslash y \rangle$

C5. [op: z ∈ {y}|K]e ⇒ if <K,z \y> then <e,z\\y>

else *nilpot*(op)

where, e.g.,

$$nilpot(+) = \begin{cases} 0 & \text{for addition} \\ nullset & \text{for union} \\ nulltuple & \text{for tuple concatenation} \\ \text{etc.} & \end{cases}$$

C6. [op: x ∈ {y ∈ S|K_1}]K_2 ⇒ [op: x ∈ S|<K_1,y \x>]K_2

C7. [+: z ∈ {y}]1 ⇒ 1

C8. {z ∈ *nullset*|K ⇒ *nullset*

C9. (∀z ∈ *nullset* | K) *BLOCK END* ∀; ⇒ ∈

where ∈ is the empty string

C10. ∃z ∈ *nullset* | K ⇒ *False*

C11. ∀z ∈ *nullset* | K ⇒ *True*

Other Cleanup Transformations

C12. ∃x ∈ (*if* c *then* e1 *else* e2)|K

⇒ *if* c *then* ∃x ∈ e1|K

else ∃x ∈ e2|K

C13. ∀x ∈ (*if* c *then* e1 *else* e2)|K

⇒ *if* c *then* ∀x ∈ e1|K

else ∀x ∈ e2|K

C14. *if* (*if* c *then* c1 *else* c2) *then*

BLOCK1 else

BLOCK2

endif

⇓

if (c & c1) *or* ¬c & c2 *then*

BLOCK1 else

BLOCK2

endif

vii. Organizing Rules into an Automatic Production System and an Efficient Implementation

This section describes a production system which can eliminate unnecessary set former, intersection, set difference, and union operations found in a program region R. The system consists of the following 5 rewrite rules which can be applied in any order exhaustively throughout R:

$\{x \in \{y \in t | K_1\} | K_2\} \Rightarrow \{w \in t | \langle K_1, y \backslash w\rangle \ \& \ \langle K_2, x \backslash w\rangle\}$
where w is a unique generated variable

$S * T \Rightarrow \{x \in S \mid x \in T\}$

$S - T \Rightarrow \{x \in S \mid x \notin T\}$

$x \in (S + T) \Rightarrow x \in S$ *or* $x \in T$ /* membership test */

$x \in \{y \in T | K\} \Rightarrow x \in T \ \& \ \langle K, y \backslash x\rangle$ " "

The above rules can be guided by an efficient 'chaining' mechanism discussed in general terms by Loveman [L1], and detailed by Kibler et al. [KI1]. In what follows we will use Kibler's approach to design an optimized production system for those rules. In addition to standard production system features such as use of a parse tree representation T of a program and a set of rewrite rules P, Kibler's system has facilities for judiciously selecting productions according to their likelihood of succeeding and for limiting the necessary range of search through T for a place where such productions can be applied. For

each node N of T, Kibler associates a locality defined by the subtree of N and an instruction stack S(N) containing a possibly empty sequence of directives; each directive has the form <location><transformation list> where <location> is either HERE or UP and <transformation list> is a list of production names. Also, for each production p of P, Kibler associates a sequence of directives of the same form as was just mentioned for stacks.

The production system begins in a starting state $[T_0,S_0,N_0]$ where T_0 is an initial tree, S_0 is a mapping from nodes of T_0 into stacks, and N_0 is a node in T_0. The transition rule which takes one state into the next is implemented by the following SETL program:

```
/*   the parse tree is represented by a set T of blank */
/* atoms along with predecessor and successor maps Tpred */
/* and Tsucc  defined on T; N is a node in T and defines */
/* the current locality; for each node n ∈ T, S(n) is a  */
/* sequence of directives; for each production p ∈ P,    */
/* prods(p) is a sequence of directives, Lhs(p) is the   */
/* Lhs pattern and Rhs(p) is the Rhs macro for p  */
  Define  Psys;
     (while N ¬= Ω)  /* halt when locality is undefined */
          (while S(N) ¬= nulltuple)
               Inst := S(N)(1);   /* select instruction */
               S(N) := S(N)(2:);  /* pop stack */
```

```
            If Inst = 'HERE' then continue;
        elseif  Inst = 'Up' then
                Top := S(N);
                S(N) := nulltuple;     /* empty stack */
                N    := Tpred(N);       /* Go up */
                S(N) := Top + S(N);
                continue;
        elseif  Temp := Dmatch(N,Lhs(Inst),Pfunc)
                     ¬= false  /* cf., Appendix E(ii) for
                                                  Dmatch */
            then N := Expand(Rhs(Inst),Pfunc);
         /*  Define new locality; cf., Appendix E(ii) for
                                                  Expand */
            S(N) := prods(inst) + S(temp);
            Replace(Temp,N);
        endif;
    end while;
    N := Tpred(N);    /* enlarge locality with stack is empty*/
  end while
end Psys;
```

In order to illustrate the previous mechanism with the five transformations presented earlier, we list these rules again but with required additional information.

NAME: P1 / structure set intersection /

ENABLING CONDITION: TYPE(S,T) = *set*, x not free in S, T.

Rule: S * T ⇒ {x ∈ S | x ∈ T}

Directions: HERE P3 P5 P4 P1 P2 UP P3 P5 P2

NAME: P2 / structure set difference /

EC: TYPE(S,T) = *set*, x not free in S, T.

Rule: S - T ⇒ {x ∈ S | x ∉ T}

Directions: HERE P3 P5 P4 P2 P1 UP P3 P5 P1

NAME: P3 / set former combinator /

EC: w not free in S, K, J

Rule: {x ∈ {y ∈ S|K}|J} ⇒ {w ∈ S|<K,y \w> & <J, x \w>}

Directions: HERE P3 UP P3 P5

NAME: P4 / union removal /

EC: TYPE(S,T) = *set*, TYPE(result) = *Boolean*

Rule: x ∈ (S + T) ⇒ x ∈ S *or* x ∈ T

Directions: HERE P4

NAME: P5 / set former removal /

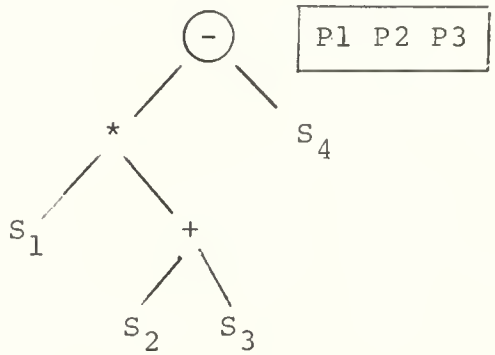
EC: TYPE(Result) = Boolean

Rule: x ∈ {y ∈ S | K} ⇒ x ∈ S & <K,y \x>

Directions: HERE P5

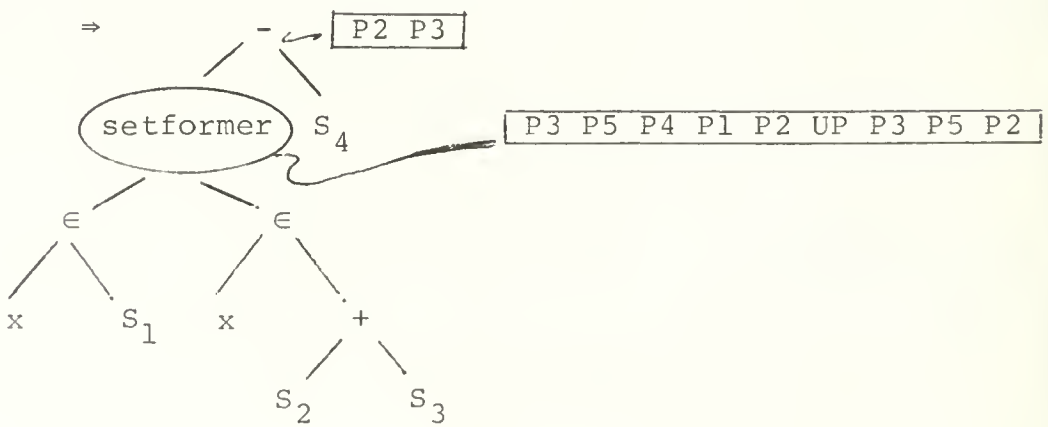
As an example, we consider a SUBSETL syntax tree for the expression $(S_1 * (S_2 + S_3)) - S_4$ and manually follow the linked transformations and changes in locality stacks. More specifically, consider the initial tree

(1)

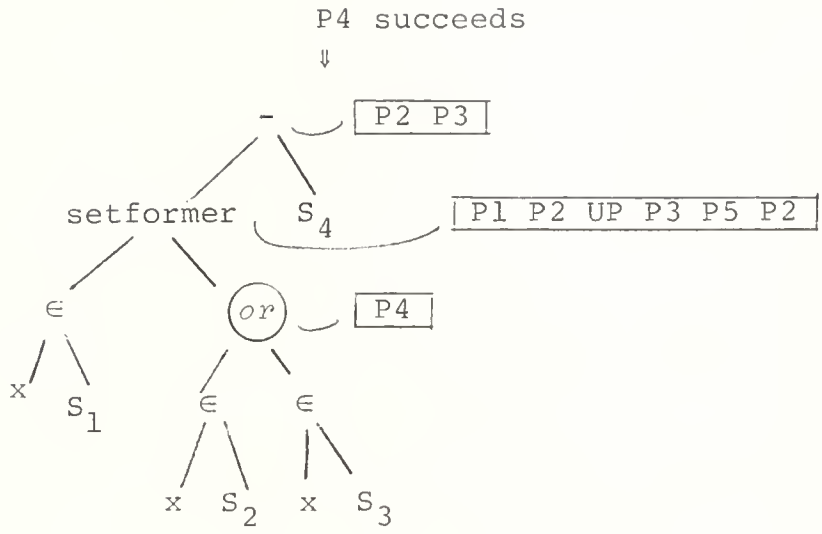


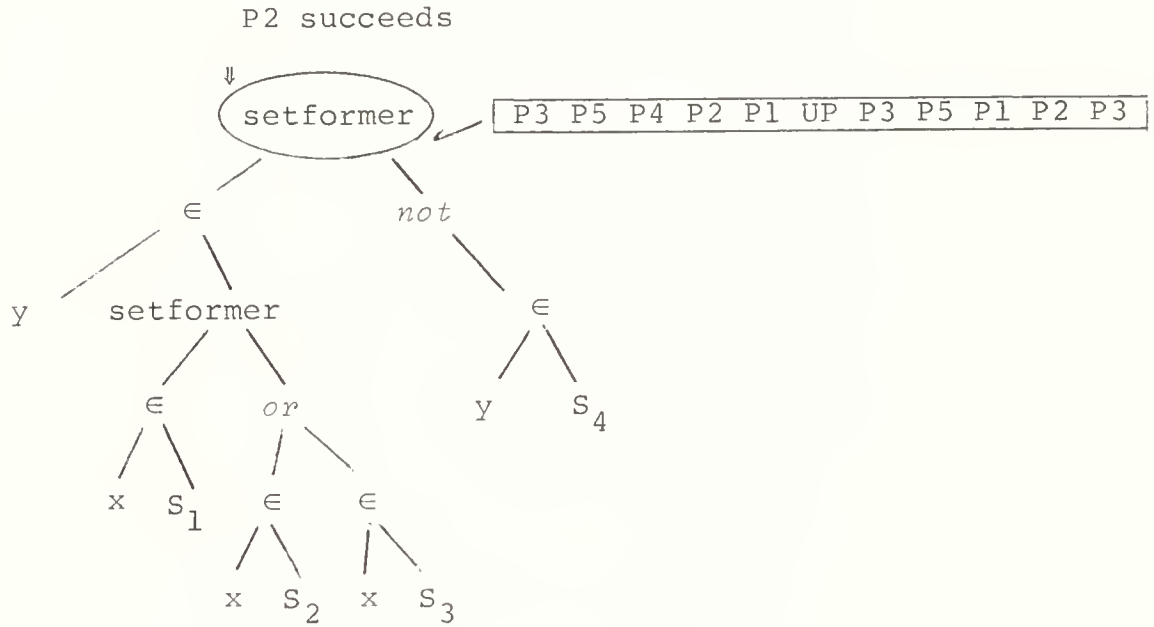
where the circled node represents the current locality, and the initial stack attached to this node contains P1 at the top and P_3 at the bottom. Processing of this tree takes place as follows:

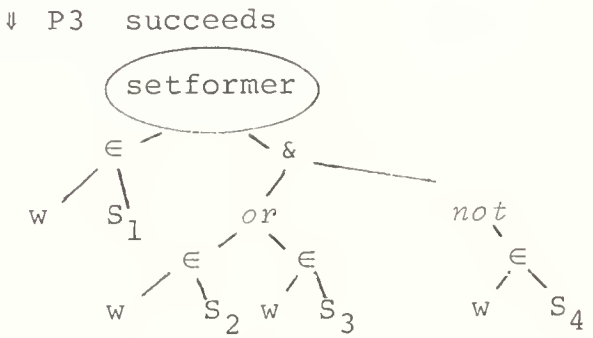
P1 succeeds



continued . . .


P4 succeeds
-
P2 P3
setformer
S4
P1 P2 UP P3 P5 P2
∈
or
P4
x
S1
∈
∈
x
S2
x
S3


P2 succeeds
setformer
P3 P5 P4 P2 P1 UP P3 P5 P1 P2 P3
∈
not
y
setformer
∈
∈
or
y
S4
x
S1
∈
∈
x
S2
x
S3


P3 succeeds
setformer
∈
&
w
S1
or
not
∈
∈
∈
w
S2
w
S3
w
S4

The locality stacks are now empty and the transformation,

$(S_1 * (S_2 + S_3)) - S_4 \overset{*}{\Rightarrow} \{w \in S_1 | (w \in S_2 \; or \; w \in S_3) \& (not \; w \in S_4)\}$

has been accomplished, only one user directive being required.

viii. A Production System for Simple Automatic Jamming

A recent paper by William Burge [BU] describes a method of loop jamming for optimizing nested recursively defined Lisp functions. All of Burge's techniques apply to corresponding iterative SETL forms, and serve to remove nesting from among set formers, tuple formers, and compound operations. An example, also considered by Burge, is to find the sum of the squares of the odd numbers of a tuple t, i.e., SUM(SQUARE(FILTER(t))) ,

where FILTER(t) = [x ∈ t: odd(x)] ,

SQUARE(t) = [x ** 2: x ∈ t] ,

and SUM(t) = [+: x ∈ t]x .

Several applications of 'jamming' transform the high level expression [+: x ∈ [y ** 2: y ∈ [z ∈ t|odd(z)]]] resulting from procedure integration into the more efficient calculation, [+: w ∈ t|odd(w)]w ** 2.

In general, this kind of transformation can be achieved automatically in a program region R by exhaustively performing the (A) productions listed below and then exhaustively performing the (B) productions.

a. Preparatory and Jamming Transformations

$[w \in t] \Rightarrow [w: w \in T]$

$\{w \in s\} \Rightarrow \{w: w \in S\}$

$[w \in t|K] \Rightarrow [w: w \in t|K]$

$\{w \in S|K\} \Rightarrow \{w: w \in S|K\}$

$[e: w \in t] \Rightarrow [e: w \in t|\mathit{TRUE}]$

$\{e: w \in S\} \Rightarrow \{e: w \in s|\mathit{TRUE}\}$

$[op: w \in s]e \Rightarrow [op: w \in s|TRUE]e$

$$[e_1: w \in [e_2: z \in t|K_2]|K_1]$$

$$\Downarrow$$

$$[<e_1, w \backslash <e_2, z \backslash y>>: y \in t|<K_2, z \backslash y> \ \& \ <K, w \backslash <e_2, z \backslash y>>]$$

where y isn't free in e_1, e_2, K_2, K_1.

$$\{e_1: w \in \{e_2: z \in s|k_2\}|k_1\}$$

$$\Downarrow$$

$$\{<e_1, w \backslash <e_2, z \backslash y>>: y \in s|<K_2, z \backslash y> \ \& \ <k_1, w \backslash <e_2, z \backslash y>>\}$$

where y isn't free in e_1, e_2, K_2, K_1

$$[op: w \in [e_2: z \in t|K_2]|K_1]e_1$$

$$\Downarrow$$

$$[op: y \in t|<K_2, z \backslash y> \ \& \ <K_1, w \backslash <e_2, z \backslash y>>]<e_1, w \backslash <e_2, z \backslash y>>$$

where y isn't free in e_1, e_2, K_2, K_1.

$$[op: w \in \{z: z \in s|K_2\}|K_1]e_1 \Rightarrow [op: y \in s|<K_2, z \backslash y> \ \& \ <K_1, w \backslash y>]<e_1, w \backslash y>$$

B. Cleanup Transformations

TRUE & K ⇒ K

K & TRUE ⇒ K

[e: x ∈ t|TRUE] ⇒ [e: x ∈ t]

{e: x ∈ s|TRUE} ⇒ {e: x ∈ s}

[op: x ∈ t|TRUE]e ⇒ [op: x ∈ t]e

[x: x ∈ t|K] ⇒ [x ∈ t|K]

{x: x ∈ s|K} ⇒ {x ∈ s|K}

{x ∈ s|TRUE} ⇒ {x ∈ s}

[x ∈ t|TRUE} ⇒ [x ∈ t]

{x ∈ s} ⇒ s

[x ∈ t] ⇒ t

ix. Named Productions for Use in Topological Sort Example of Chapter 3.

NAME: ∃FORMAT

Rule: ∃ x ∈ s|K ⇒ ∃ x ∈ {u ∈ s|<K,x \u>}

NAME: SETEQNL

Rule: s = *nullset* ⇒ ([+: u ∈ s]1) *eq* 0

where u is a unique name in TOPSORT's namespace, x, s, K are patterns, and <,\ ,> are metasymbols used to denote substitution.

NAME: {To*

Rule: {z ∈ s|z = y} ⇒ {y} * s
where z isn't free in y

NAME: *SIMP

Rule: A * B ⇒ A

ENABLING PRED: B *INCS* A

NAME: {= NL

Rule: {x ∈ s|K} ⇒ *nullset*

ENABLING PRED: ∀x ∈ s|*not* K

NAME: IDEM

Rule: A + *nullset* ⇒ A

NAME: USELESS=

Rule: A = A; ⇒ ε where ε is the empty string

NAME: EMPTYELSE

Rule: *if* P *THEN* B *ELSE* *ENDIF*;

⇒ *if* P *THEN* B *ENDIF*;

NAME: {TOIF

Rule: {z ∈ {y}|K(z)} ⇒ *if* K(y) *then* {y} *else nullset*

where z is not within the scope of a bound variable y in K.

NAME: DISTIF1

Rule: A = B $\pm$ if P then x else y;

⇒ if P then A = B $\pm$ x; else A = B $\pm$ y; endif;

where A cannot occur free in P

NAME: {to∀

Rule: A = {x ∈ s|K} ⇒ (∀x ∈ s|K) A = A + {x}

where A is not free in S or K

NAME: ∀SIMP*

Rule: (∀x ∈ s|x ∈ T) BLOCK ⇒ (∀x ∈ S * T) BLOCK

NAME: *COMMUTE

Rule: A * B ⇒ B * A

where B, A are disjoint

NAME: ∀CONC

Rule: $(\forall x_1 \in s_1)(\forall x_2 \in s_2|K)$ BLOCK

⇒ $(x_1 \in s_1, x_2 \in s_2|K)$ BLOCK

NAME: ∀COMMUTE

Rule: $(\forall x_1 \in s_1, x_2 \in s_2|K)$BLOCK

⇒ $(\forall x_2 \in s_2, x_1 \in s_1|K)$ BLOCK

where s_1, x_1 are disjoint with x_2, s_2 & block is order independent.

NAME: JAM

Rule: $(\forall x \in s)\ BLOCK_1$ end $\forall$; $(\forall y \in s)\ BLOCK_2$ end $\forall$
$\Rightarrow (\forall w \in s)\langle BLOCK_1, x \setminus w\rangle;\langle BLOCK_2, y \setminus w\rangle$ end $\forall$

ENABLE PRED: where $BLOCK_1$ and $BLOCK_2$ are disjoint and w is a unique generated name.

NAME: ∀BRKUP

Rule: $(\forall x_1 \in s_1,\ x_2 \in s_2 | K)$ BLOCK
$\Rightarrow (\forall x_1 \in s_1)(\forall x_2 \in s_2 | K)$ BLOCK

NAME: DEADELIM FORMAT(SCOPE = L)
'Remove dead code from Region L'

Rule: EXECUTE(DEADELIM)
'Rule executes a procedure'

NAME: VSUBST FORMAT(SCOPE = S)

Rule: EXECUTE(VSUSBST)

VSUBST substitutes the expression e to the right of the first matched assignment statement A for occurrences of the variable name V (to the left of that assignment) in statement number s where s is a user supplied parameter. The value of the ud map applied to occurrences of V in s must equal the program point p containing A. Furthermore all paths from p to s must be clear of definitions to variables of e.

Appendix E. Assorted Utility Routines for a Source to Source Transformational Implementation

i. The Unparser

The Unparser procedure shown below is used to print the source code of a SUBSETL program, given the program in parse tree form. We assume that the parse tree is based on the grammar given in Appendix A and is generated by any suitable parser (for which we omit any further description). The tree will consist of a set N of nodes (each node implemented as a blank atom) and a map Tsucc associating each node n with an ordered tuple Tsucc(n) of successor nodes. Aside from Tsucc, we also make use of a number of other maps defined on N. These include the following.

1. Leaf(n) is *true* when the node n is a leaf and *false* otherwise.
2. Label(n) will be a token value if n is a leaf, or the lexical type (e.g. 'block', 'statement') associated with an internal node n.
3. Number(n) represents the statement number of node n when Label(n) = 'statement'.

The following is a SETL version of Unparser:

```
/* Program is the root node of the parse  tree */
 Define Unparser(Program);
   /* Initialize global variables */
   line := nullchar; /* line is line buffer */
```

```
   Columns:= 60;  /* max no. of chars  for line */
   Margin := 10;  /* source line begins after column 9 */
   Indent := 0
   Printpart(Program);  /* begin unparsing */
   Flush;               /* print last line */
end;

/* unparse the block represented by parameter Block */
Define Printblock(Block);
   Indent := Indent+1;  /* indent statements of block */
   (∀State ∈ Tsucc(Block))
      Printstate(State);   /* print block statements */
   end ∀;
   Indent := Indent - 1;   /* restore previous indentation*/
end;
/* unparse the statement State */
Define Prinstate(State);
   Flush;            /* print current buffer line */
   Putleaf(number(State)); /* add statement no. to line*/
   Tab(Margin + Indent*2); /* begin  statement in
                                          correct col */
   (∀n ∈ Tsucc(State))
      Printpart(n);  /* unparse the parts of statement */
   end ∀;
end;
```

```
/* Print the subpart node of a statement */
Define printpart(node);
   if Label(node) = 'BLOCK' then
      Printblock(node);      else
   if Leaf(node)  then
      Putleaf(Label(node)).  /* print token */
      Tab(#Line+2);  /* add 2 spaces after each token */
                     else
      (∀x ∈ Tsucc(node))     /* print subparts of node */
         Printpart(x);
      end ∀;
   endif;
end;
/* Add the token string to the source line */
Define Putleaf(string);
   /* if string cannot fit on line then print line and */
   /* add string to next line which is indented */
   if #line + #string > Columns then
      Flush;
      Tab(Margin + Indent * 2 + 2);
   endif;
   Put(string);
end;
/* Add spaces to line up to the column Col */
Define Tab(Col);
   line := line + [+: #line < i ≤ Col]' ';
end;
```

```
/* print line and initialize next line */
Define  flush;
   if #line > 0  then
      Print(line);
   endif;
   line :=nullchar;
end;
```

ii. Pattern Matcher

Dmatch is a function subprogram which attempts to match a pattern, Pat, to a subtree within a SUBSETL parse tree whose root node we call Tree, using a depth first search through Tree. The parse tree structure is as described in connection with the Unparser algorithm. Patterns will have the same structure as trees. Maps Pleaf, Plabel, and Psucc will be defined on the nodes of a pattern tree and serve much the same purpose as Leaf, label, and Tsucc respectively. Moreover, if n is a leaf of a pattern tree, then application of the boolean valued function Literal(n) will yield *true* when Plabel(n) is a literal and *false* when it is a pattern variable. Dmatch will return the root of the subtree within Tree at which a first successful match occurs. When matching succeeds, Dmatch will also return the map Pfunc as a parameter. This map associates each pattern variable x of Pat with the root Pfunc(x) of a

subtree of Tree matched by x. Note that Pfunc is useful for macro expansion subsequent to matching (cf. the Expand algorithm in this appendix, section iii). If Dmatch is unable to find a successful match, then it will return *false*.

The actions of the pattern matching routine Match which is invoked by Dmatch has been described in detail in Chapter 3 (B).

The following SETL code implements Dmatch, Match and various related auxiliary routines.

```
/* Tree and Pat are roots of a parse tree */
/* and pattern tree respectively */
definef Dmatch(Tree,Pat,Pfunc);
   if  Match(Tree, Pat, Pfunc)  then
      Return Tree;  else /* return node where matching
                                              succeeds  */
   if ∃x ∈ Tsucc(Tree)|Subtree := Dmatch(x,Pat,Pfunc) then
      Return Subtree;  else
      Return false;
   endif;
end;
/* Match is a routine whose sole purpose is to initialize
   the pattern variable map Pfunc. This permits the main
   match procedure Matchl  to call itself recursively
   and preserve the previous value of Pfunc */
definef Match(Tree,Pattern,Pfunc);
   Pfunc :=  nullset;
   Return  Matchl (Tree, Pattern, Pfunc);
end;
```

```
/* Matchl  attempts to match the parse tree whose
   root node is Tree by the pattern tree whose root
   node is Pattern. If the match succeeds, for each
   pattern variable x in the pattern tree, Pfunc(x)
   will be the node in the parse tree matching x. */
Definef Matchl(Tree,Pattern,Pfunc);
   if Pleaf(Pattern) then
      if Literal(Pattern) then
         Return Plabel(Pattern) = Label(Tree); else
      if Plabel(Pattern) ∉ Dom Pfunc then
         Pfunc(Plabel(Pattern)) := Tree;
         Return true; else
         Return Equals(Pfunc(Pattern), Tree);
      endif; else
   if #Psucc(Pattern) = #Tsucc(Tree) then
      Return ∀ 1 ≤ n ≤ #Psucc(Pattern)|
            Matchl(Tsucc(Tree)(n),Psucc(Pattern)(n),Pfunc);
                                                    else
      Return false;
   endif;
end;
/* Equals is a predicate that decides whether 2 trees
   T1 and T2 have the same values; i.e., the same
   structure and the same leaf values */
Definef Equals(T1,T2);
   if Leaf(T1) then
      Return Label(T1) = Label(T2); else
```

```
        Return ∀ 1 ≤ i ≤ #Tsucc(T1) |
                Equals(Tsucc(T1)(i),Tsucc(T2)(i));
    endif;
end;
```

iii. Macro Expander

The function subprogram Expand performs macro expansion by generating a tree from a pattern Pat and a pattern variable map Pfunc. It replaces each pattern variable x within Pat by the tree Pfunc(x). The tree generated will then have the same structure as Pat down to the leaves of Pat.

A utility routine copytree is used to make a fresh copy of a tree.

In SETL these substitution routines are as follows:

```
Definef Expand(Pat,Pfunc);
    if Pleaf(Pat)& ¬Literal(Pat) then
        Root := Copytree(Pfunc(Plabel(Pat))); else
    if Leaf(Pat) then
        Root := newat;
        Label(Root):=  Plabel(Pat); else
        Root := newat;
        Tsucc(Root) := [Expand(x): x ∈ Psucc(Pat)];
    endif;
    Return Root;
end;
```

```
    Definef Copytree(Tree);
        node := newat;    Label(node) := Label(Tree);
        Tsucc(node) := [Copytree(x):    x ∈ Tsucc(Tree)];
        Return node;
    end;
```

iv. Utility Routines for Formal Differentiation

The two procedures, Regconst and Indvars, shown below compute the region constants and induction variables within a loop L. Both these routines can be used (with minor adjustments) in all of the FD frameworks discussed in Chapters III and IV. They make use of the following global variables:

1. L, Leaf, Label, Tsucc represent the parse tree for the loop L.
2. Vars is the set of variable names used in L.
3. For each v ∈ Vars, Defs(v) is the set of definition points in L for v.
4. F and D are the elementary form and derivative tables.
5. Pleaf, Pvar, Pleaves and Plabel are maps defined for patterns (cf. Appendix E(ii)).

Regconst defines the global variable RC, the set of all nodes n ∈ L in which Text(n) is a region constant expression. Indvars computes the global map IV which associates the set of induction variables IV(x,f) for each pattern variable x within each elementary form f ∈ F.

```
/* RC is the set of   nodes n ∈ L such that Text(n) is a
   region constant expression */
Define Regconst;
     /* initialize RC to the set of leaf nodes corresponding
        to constant and region constant variables */
     RC := {n ∈ L|Leaf(n) & (Label(n) ∉ Vars or
                          Defs(Label(n)) = nullset)};
     /* Find the region constant expressions */
     (while ∃ n ∈ (L-RC)|(∀y ∈ Tsucc(n)|y ∈ RC))
                          RC := RC + {n};
     end while;
end;
/* Compute the set IV(x,f) of induction variables for every
   pattern variable x in each elementary form f ∈ F */
Define Indvars;
   /* compute the set of region constant variables */
   Rvars := {Label(n): n∈RC|Leaf(n) & Label(n) ∈ Vars};
   /* Initialize IV */
   IV := nullset;
   (∀f ∈ F, x ∈ Pleaves(f)|Pvar(x))
      IV(Plabel(x),f) := Vars - Rvars;
   end ∀;
```

```
   /* Find each IV set by a negative transitive closure
      algorithm */
   (while ∃[x,f] ∈ Project(IV,2), v ∈ IV(x,f), n ∈ Defs(v)
       | ¬(∃t ∈ D(x,f) |Match(n,t(1),Pfunc) & t(4)))
   /* t(4) is a SETL code block whose value must be true
      for matching to succeed */
      IV(x,f) := IV(x,f) - {v};
   end while;
end;

/* Compute the set of leaf nodes of pattern Pat */
Definef Pleaves(Pat);
   if Pleaf(Pat) then
      Return {Pat}; else
      Return [+: x ∈ Psucc(Pat)] Pleaves(x);
   endif;
end;
```

The following routine, Postorder, computes a tuple containing the nodes of a parse tree (of the standard form that we have been using all along) arranged in postorder, i.e., left to right successors before root order.

```
/* Traverse successors from left to right.*/
   Then visit the root */
Definef Postorder(Tree);
   T := nulltuple;
   Post(Tree,T);
   Return T;
end;

Define  Post(Tree,T);
   if Leaf(Tree)  then return;
   endif;
   (∀x ∈ Tsucc(Tree)|¬Leaf(x))
      Post(x,T);
   end ∀;
   T := T + [Tree];
end;
```

v. Revised Matching and Expansion Routines

The two routines, Match and Expand shown below are more powerful versions of the simpler routines of the same name described earlier in this appendix. Match accepts as input parameters the root nodes Tree and Pattern of a parse tree and pattern tree respectively. Match initializes Pfunc to the empty set and calls the main matching utility Matchl which defines the pattern variable map Pfunc. However, the parameters Tree and Pattern for Matchl must be tuples whose components are the root nodes of parse and pattern subtrees. Likewise, the parameter Pattern used by Expand will be a tuple of root nodes of subpattern trees.

These routines are sufficiently powerful to handle patterns specified in the FD tables of Appendix C (iii). Thus, they are fundamental components of the FD implementation design of Chapter 4.

```
/* Match initializes Pfunc to nullset and passes the root
   nodes Tree and Pattern to Match 1 */
Definef Match(Tree,Pattern, Pfunc);
   Pfunc := nullset;
   Return  Matchl ([Tree], [Pattern], Pfunc);
end;

/* Match 1 attempts to  match an ordered forest 'Pattern' */
/* to an ordered parse tree forest 'Tree'. During successful */
/* matching Pfunc will be built up by associating  pattern   */
```

```
/* variables x with parse tree nodes Pfunc(x) matched by x. */
/* When matching fails, Pfunc will be restored to its
   previous state */
Definef Matchl(Tree, Pattern, Pfunc);
   Local Savfunc;
   if Pattern = nulltuple  then /* consider trivial match- */
      if Tree = nulltuple  then /* int decisions first     */
         Return True; else
         Return False;
      endif;
   endif;
   P := Pattern(1);  /* fetch the first pattern */
   if Tree = nulltuple & ¬Control(P)  then
      return false;
   endif;
   if Control(P) then    /* if pattern is a procedure */
      if Plfabel(P)(P)  then /* execute Plabel(P)(P)  */
         Return Matchl(Tree,Pattern(2:),Pfunc)  else
         Return false;
      endif;
   endif;
   T := Tree(1) ;     /* fetch first parse tree from forest */
   if Pvar(P) then   /* if p is a pattern variable */
      if ¬Pvarut(P,T) then  /* execute a utility routine */
         return false;  /* which handles this case; if */
      endif          /* False is returned matching fails; */
   endif;    /* else, Pvarut will associate Pfunc(Plabel(P))
                                                with T */
```

```
if Alt(P) then    /* if P represents alternation save          */
   Savfunc := Pfunc; /* Pfunc and match one of the alternands.*/
   (∀x ∈ Psucc(P))
      if Matchl(Tree, x + Pattern(2:) then
         return true; else /* return true if success */
         Pfunc := Savfunc; /* restore Pfunc, and try again; */
      endif;               /* otherwise return false   */
   end ∀;                  /* if no alternand succeeds */
   return false;
elseif Literal(P)  then
   return Plabel(P) = Label(T) &
        Matchl(Tree(2:), Pattern(2:), Pfunc);
elseif Leaf(P)  then
   return Matchl(Tree(2:), Pattern(2:), Pfunc); else
   return Matchl(Tsucc(T), Psucc(P),Pfunc) &
       Matchl(Tree(2:), Pattern(2:), Pfunc);
endif;
end;
/* Pvarut  handles  pattern  variables. The parameters
   P and T are the roots of a pattern and parse tree resp. */
Definef Pvarut(P,T);
   Pname := Plabel(P); /* The value of Pname is the name of */
       /* the    pattern variable at P */
   if Pgen(P) then     /* if a generated name for the */
       /* assignment variable is required */
```

```
      Sname := '#' + Pname; /* fetch special name to */
      Pfunc(Sname) := Pfunc(Sname)+1 ort 1; /* increment counter
      Pname := Pname+Code(Pfunc(Sname)); /* generate  name */
endif;
if Pname ∉ DOM Pfunc then   /* if pattern    variable has not */
   Pfunc(Pname) := T;  /* been  previously encountered record
   return true; else;  /* it; else, check for consistency. */
   return Equals(Pfunc(Pname,T)); /* equals is the same as in */
   endif;              /* connection  with the earlier Match */
end;                   /* routine in setcion ii. */
/* Pattern is an ordered forest of patterns. */
/* Expand returns an ordered forest of parse trees. */
Definef Expand(Pattern, Pfunc);
Local  Savfunc;
if Pattern =  nulltuple  then
   return nulltuple;   /* trivial expansion */
endif;
P := Pattern(1);       /* Fetch first pattern. */
if Control(P) then     /* If pattern is a procedure,execute */
   if Plabel(P)(P)then/* Plabel(P)(P) */
      return  Expand(Pattern(2:), Pfunc) else
      return false;
   endif;
elseif Alt(P) then     /*    If P represents alternation */
   Savfunc := Pfunc;   /*  save Pfunc and try to expand */
   (∀x ∈ Psucc(P))     /* an alternand successfully */
      if Temp := Expand(x+Pattern(2:)) ≠ False then
         return Temp; else
```

```
            Pfunc := Savfunc; /* restore Pfunc after
                                 each failure */
         endif;
      end ∀;
      return false;
   elseif Literal(P)  then
      Root := newat;
      Label(Root) := Rlabel(P);
   elseif ¬Leaf(P)  then
      Root := newat;
      if Temp := Expand(Psucc(P), Pfunc) ≠ false then
         Tsucc(Root) := Temp; else
         return false;
      endif;
   endif;
   if Pvar(P)  then   /* If P must be assigned fetch the name */
      Pname := Plabel(P);  /* of the pattern    variable. */
      if Pgen(P)  then  /* If the name is generated, create new
                                                          name */
         Sname := '$'+Pname;
         Pfunc(Sname) := Pfunc(Sname) + 1 ort 1;
         Pname := Pname + Code(Pfunc(Sname));
      endif;
      if Ngen(P)  then  /* In the case a tree is stored in Pfunc */
         if Leaf(P) & ¬Literal(P) then
            Root := newat;         /* create tree */
            Label(Root) := newname;  /* create new variable name */
         endif;
         Pfunc(Pname) := Root;       /* store tree */
```

```
      endif;
      if Pname ∉ Dom Pfunc then  /* If Pfunc cannot satisfy */
         return false;          */ Pname, expansion fails */
      endif;
endif;
if Temp := Expand(Pattern(2:), Pfunc) ≠ false then
   return[Root] + Temp;  else
   return false;
endif;
```

It would be convenient for our FD implementation to include a compiler for translating the patterns given in Appendix C (iii) into their tree representation — the form which must be passed as input to Match and Expand. Until such a compiler is built, however, we will have to perform this translation by hand coding in SETL. In the following discussion we give rules for translating pattern expressions into pattern trees (cf. the informal discussion of patterns in Chapter 4 (B) and the formal syntactic description of our pattern language in Appendix C (iii)).

Patterns will be implemented using a set N of blank atoms (representing nodes), a successor map Psucc(n), a root node r, and an assortment of maps defined on N. Some of these maps, especially Plabel, Pleaf and Literal are used in much the same way as before.

To synthesize a pattern tree from a pattern expression we will use the following order of evaluation.

1. Process pattern expressions from within innermost to outerpost brackets (these include parentheses and predecessor formation brackets). Evaluate subexpressions in the order defined by the following rule.

2. A pattern expression e selected by step 1 is processed by evaluating all concatenations before evaluating alternations. Next, if e is enclosed within parentheses, we evaluate the factor (e) as the value of e; otherwise, if e is the argument of a predecessor formation operation evaluate the term [e] according to rule 6 below.

The actual pattern construction steps which must be taken to transform pattern expressions into pattern trees (using the preceding order of evaluation) are as follows:

1. The value of a literal symbol; e.g., ',', is a tuple [n := *newat*]. On encountering such a symbol, we also execute the following assignments:

```
Leaf(n) := Literal(n) := true;
Plabel(n) := ',';
```

2. The value of a procedure name; e.g., !Cvar is a tuple [n := *newat*]. In processing such a name, we also perform the following actions.

```
Leaf(n) := Control(n) := true;
Plabel(n) := Cvar; /* Cvar is a procedure name */
```

3. The value of a pattern variable x is [n := *newat*]. In processing such a variable we execute the code

```
Leaf(n) := Pvar(n) := True;
Plabel(n) := 'x';
```

If the pattern variable is dotted (e.g., x.) we will also execute

```
Pgen(n) := True;
```

For pattern variables x* we must perform the additional assignment,

```
Ngen(n) := true;
```

4. The value of a pattern name C is the value of the pattern expression on the right side of the assignment which defines C. This value will always be a tuple of one or more components each of which is a blank atom representing the root node of a subpattern tree.

5. If P1 and P2 are two pattern expressions then the value of the concatenation of P1 with P2, written P1 P2 as in Snobol, is the SETL tuple concatenation P1' + P2' where P1' is the value of P1 and P2' is the value of P2.

6. If P is a pattern expression and P' is the value of P then the value of the predecessor formation [P] is [n := *newat*], in connection with which we perform Psucc(n) := P'.

7. If P1,P2,...,PN are pattern expressions whose values are P1',P2',...,PN' then the value of the alternation P1|P2|...|PN is [n := newat] in connection with which we execute the code

```
Alt(n) := true;
Psucc(n) := [P1',P2',...,PN'];
```

Note that for recursive pattern definitions; e.g.,

```
(1)        Params = q3. ',' Params | q3.
```

where the pattern name Params appears on the right and left hand side of the same assignment, we assign [*newat*] which is the value of the alternation expression on the right hand side of (1) to the pattern name Params. After doing this, we can evaluate the two alternands. The actual SETL code used to evaluate (1) is given below.

```
/* evaluate q3. ',' Params */
T1 := [n := newat];   /* evaluate q3. */
Pgen(n) := Pvar(n) := true;
Plabel(n) := 'q3';
T1 := T1+[n := newat];;  /* evaluate q3. ',' */
Pliteral(n):= true;
Plabel(n) := ',';
T1 := T1+Params := [save := newat];
                    /* evaluate q3. ',' Params */
                    /* evaluate q3. on the right */
T2 := [n := newat];
Pgen(n) := Pvar(n) := true;
Plabel(n) := 'q3';
/* evaluate alternation */
Alt(save) := true;
Psucc(save) := [T1,T2];
```

APPENDIX F. ADDITIONAL CASE STUDIES

In the following Appendix, we explore the potential of formal differentiation for algorithm optimization by studying four more programs. These programs are somewhat more complicated than those studied in Chapter 4, and they require minor adjustments and extensions to the transformations found in Appendix C (iii). These extensions to the F, D, and Init tables point to further extensions which would lead to a fairly complete implementation design of FD for SETL.

The first example considered is a derivation of an efficient bubble sort algorithm. (Note that a similar derivation was first described in [Sch 9, Sch 10].) In its base form, the bubble sort can be written in SETL as follows.

```
(1)  1      (while ∃ 1 ≤ n < #v|v(n) > v(n + 1))
     2         [v(n), v(n+1)] := [v(n+1), v(n)];
     3      end while;
```

where the input variable v is a tuple of integers to be sorted in place.

In order to apply FD to (1) we must first recognize the existential quantifier within (1) as an instance of the general form (18) of Chapter II (c). For this to be the case, we must demand that the predicate v(n) > v(n+1)

occurring in line 1 of (1) should be independent of #v. But this is obvious since #v is a region constant expression; i.e., by simple range analysis of the variable n we may know that the double indexed assignment to v at line 2 of (1) will not spoil the value of #v. Thus, we can prepare (1) for FD by transforming the expression $\exists\, 1 \leq n < \#v \mid v(n) > v(n+1)$ into $n := [min: 1 \leq m < \#v \mid v(m) > v(m+1)]m \neq \Omega$.

After this, we can improve the bubble sort by reducing

(2) $n = [min: 1 \leq m < \#v \mid v(m) > v(m+1)]m$.

The derivative code for (2) relative to the changes to v at line 2 of (1) is

```
(3)  T := if 1 ≤ n & n < n then [n] else nulltuple
          +if 1 ≤ n-1 & n-1 < n then [n-1] else nulltuple
          +if 1 ≤ n+1 & n+1 < n then [n+1] else nulltuple
          +if 1 ≤ n & n < n then [n] else nulltuple;
     [v(n), v(n+1)]  :=  [v(n+1), v(n)];
     if ∃ m ∈ T | v(m) > v(m+1) then
            n := m; else
     if ¬(v(n) > v(n+1)) then
            n := [min: n+1 ≤ m < #v|v(m) > v(m+1)]m;
     endif;
```

which is obtained from a general update rule (for multiple indexed assignments) which combines the efficient Rule 2 Technique (21) of Chapter II (c) with the derivative code (9)

of Chapter II (c). Transformation is completed by inserting an assignment (2) at the entry to the while loop within (1).

The actual changes to the FD implementation of Chapter 4 necessary to perform the transformations just described involve rather straightforward additions to the F, D, and Init tables and a minor generalization of Rule 2. We consider these extensions to be attractive future possibilities.

To obtain a final form of the bubble sort, several low level cleanup transformations must be applied. Specifically, the assignment to T within the derivative code (3) can be simplified if we note that both the relations n < n and n + 1 < n can be replaced by *False*, while the relation n - 1 < n can be replaced by *True*. Further applications of simplifying syntactic transformations of the sort found in [ST2] and Appendix D lead to a more attractive assignment, T := *if* 2 $\leq$ n *then* [n-1] *else* *nulltuple*. Still further simplification of (3) is possible if we note that the relation v(n) > v(n+1) (which occurs in the *while* loop predicate) holds just prior to the multiple assignment [v(n), v(n+1)] := [v(n+1), v(n)] but its negation holds immediately afterwards. Consequently, the IF statement occurring in (3) can be optimized into the following form.

```
    if  m ∈ T|v(m) > v(m+1) then
            n := m;  else
            n := [min: n+1 ≤ m < #v|v(m) > v(m+1)]m;
    endif;
```

The changes to (1) made thus far lead to the following code.

```
(4)  1      n := [min: 1 ≤ m < #v|v(m) > v(m+1)]m;
     2      (while n ≠ Ω)
     3           T := if 2 ≤ n then [n-1] else nulltuple;
     4           [v(n), v(n+1)] := [v(n+1), v(n)];
     5           if ∃m ∈ T|v(m) > v(m+1) then
     6                n := m;  else
     7                n := [min: n+1 ≤ m < #v|v(m) > v(m+1)]m;
     8       endif;
```

One last chain of cleanup transformations will bring us to our goal. First we apply the directive VSUBST, 3 (cf. Appendix D (IX)) which replaces the single use of T at line 5 by the conditional expression at line 3, and then deletes line 3. Next we make successive applications of transformations C12, C10, C3, and C14 of Appendix D (VI) followed by standard simplifying boolean identities of Appendix D(ii) to the IF statement at line 5 of (4). Our final form of the bubble sort is then

```
(5)  n := [min: 1 ≤ m < #v|v(m) > v(m+1)]m;
     (while n ≠ Ω)
          [v(n), v(n+1)] := [v(n+1), v(n)];
          if 2 ≤ n & v(n-1) > v(n) then
               n := n-1;  else
               n := [min: n+1 ≤ m < #v|v(m) > v(m+1)]m;
          endif;
     end while;
```

The manual effort required to apply all of the cleanup transformations mentioned seems exorbitant. It seems likely, however, that efficient production systems of the kind proposed by Kibler and Standish [KI1] and exemplified within Appendix D (VII) and (VIII) might reduce the amount of manual intervention. Such transformation families might be enabled by assertions propagated from loop predicates throughout the program text, and applied automatically. Methods of this kind tailored to SETL have been worked out by E. Deak [D]. Further research along these lines look promising for the future.

For another example of algorithm improvement by FD, we consider an algorithm which finds all nonterminals in a context free grammar from which the empty string Λ can be derived. The base form SETL program we use to specify this algorithm accepts the grammar G as input, and outputs the appropriate set S of nonterminals. G is represented as a function which maps each nonterminal n

into a set of terms G(n) immediately derived from n. Each term t ∈ G(n) is a tuple; each component of t contains either a terminal or nonterminal of G.

We begin our consideration of this example with the following succinct program,

```
(6)  1        S := {n ∈ Dom G | Λ ∈ G(n)};
     2        (while ∃ n ∈ Dom G|n ∉ S & (∃t∈G(n)|(∀y∈t|y∈S)))
     3             S := S + {n};
     4        end while;
```

Next we prepare the *while* loop predicate of (6) for FD by applying the following transformations (all of which are described in Appendix D (v): P3 and P13 to the universal quantifier, P2 and P13 to the inner existential quantifier of the predicate, and finally P6 to the outermost quantifier. The predicate which results is

```
(7)  ∃n ∈ {x ∈ Dom G | x ∉ S & ([+: t ∈ {y ∈ G(x) |
                ([+: z ∈ {w ∈ y|w ∉ S}]1 = 0)}]1 ≠ 0)}
```

To reduce the outermost setformer appearing in (7), we can use a single directive:

```
(8)  $FD,2,W = {x ∈ Dom G|x ∉ S & ([+: t ∈ {y ∈ G(x) |
                ([+: z ∈ {w ∈ y|w ∉ s}]1 = 0)}]1 ≠ 0)}.
```

However, to handle (8) the FD implementation design of Chapter 4 must incorporate a few revisions. This is because

our current version of algorithm lSETL will fail to recognize that $c_1(y) = \{w \in y | w \notin s\}$ is reducible. In fact lSETL excludes from reduction any expression which, like c_1, has set or tuple valued discontinuities. However, when the range of values for a set or tuple valued discontinuity y can be bounded explicitly; e.g., when y belongs to a set valued variable which is invariant within the optimization loop, we can relax these restrictions. In the case of c_1, the usetodef map will help determine that the value of the tuple valued discontinuity y used in c_1 must belong to the range of the map G. This fact allows us to define c_1 on entrance to the *while* loop L of (6) by executing

```
(9)  (∀x ∈ Dom G, y ∈ G(x))
          c1(y) := {z ∈ y|z ∉ s};
     end ∀;
```

and to keep c_1 available in L by executing prederivative code

```
(10) (∀y ∈ ({n}-s), t∈{w ∈ Dom c1|y ∈ w})
          c1(t) := c1(t) - {y};
     end ∀;
```

just prior to the definition s := s + {n} at line 3 of (6).

Assuming then that algorithm lSETL can mark c_1 reducible, it will also be able to recognize the other reducible subexpressions of W (cf. (8)). These are

$$c_2(y) = [+: z \in c_1(y)]1 ,$$
$$c_3(x) = \{y \in G(x) \mid c_2(y) = 0\} ,$$
$$c_4(x) = [+: t \in c_3(x)]1 , \text{ and}$$
$$c_5 = \{x \in \mathit{Dom}\ G \mid x \notin s \ \&\ c_4(x) \neq 0\}.$$

Next we envision a slightly more refined version of the reduction algorithm 2SETL of Chapter 4 in which successive applications of reduction would interleave the application of a standard collection of cleanup transformations. This would enable us to simplify (10) directly into

(10')
```
(∀t ∈ {w ∈ Dom c1 | n ∈ w})
     c1(t) := c1(t) - {n};
end ∀;
```

The differentiation rule for c_1 will require that the calculation $c_6(n) = \{w \in \mathit{Dom}\ c_1 \mid n \in w\}$ occurring within (10') should be reduced. c_6 can be treated as a special case of elementary form 1 with conjunct H described in Appendix C (iii). Since c_6 is invariant within L we only need to initialize by executing

(11)
```
c6 := nullset;
(∀x ∈ Dom c1, y ∈ x)
     if y ∈ Dom c6 then
          c6(y) := c6(y) + {x}; else
          c6(y) := {x};
     endif;
end ∀;
```

immediately after executing (9). Consequently, the code (10') further simplifes to

(12)
```
(∀t ∈ Ntinrhs(n))
    c1(t) := c1(t) - {n};
end ∀;
```

where we suppose that Ntinrhs is the user supplied name replacing c_6. The reduction algorithm can now proceed straightforwardly from inner to outer expressions in a manner consistent with the method of Chapter 4 and the tables of Appendix C (iii). The prederivative of c_2 with respect to the change to c_1 occurring in (12) is

(13)
```
c2(t) := c2(t) - [+: y ∈ {n}]1;
```

which simplifies immediately to

(13')
```
c2(t) := c2(t) - 1;
```

Since no uses of c_1 occur in the derivative code for c_2, we can define c_2 at the entry to L and remove all dead definitions to c_1 from the program. This puts the original program (6) into the following transitional form:

```
(14) 1        s := {n ∈ Dom G | Λ ∈ G(n)};
             /* prologue */
     2       (∀x ∈ Dom G, y ∈ G(x))
     3            c2(y) := [+: z ∈ y |z ∉ s]1;
     4       end ∀;
     5       Ntinrhs := nullset;
     6       (∀x ∈ Dom c2, y ∈ x)
     7            if y ∈ Dom Ntinrhs then
     8                 Ntinrhs(y) := Ntinrhs(y)+{x}; else
     9                 Ntinrhs(y) := {x};
     10           endif;
     11      end ∀;
             /* main loop */
     12      (while ∃n ∈ {x ∈ Dom G|x ∉ s & ([+: t∈{y∈G(x)|
                       (c2(y) = 0)}]1 ≠ 0})
     13           (∀t ∈ Ntinrhs(n))
     14                c2(t) := c2(t) - 1;
     15           end ∀;
     16           s := s + {n};
     17      end while;
```

Next we reduce c_3; its prederivative relative to the change to c_2 at line 14 of (14) is

```
(15)        if c2(t) = 1 then
                (∀q ∈ {w ∈ Dom G|t ∈ G(w)})
                    c3(q) := c3(q) + {t}; /* strict set
                                             addition */
                end ∀;
            endif;
```

in which $c_7(t) = \{w \in Dom\ G \mid t \in G(w)\}$ must be reduced. Since a use of c_2 occurs within (15), c_2 cannot be eliminated. Thus, the system will request the user to supply a name for c_2. (Let this be Noncnt.) The initializing code for c_3 inserted just after line 11 of (14) is

```
(16)        c3 := nullset;
            (∀x ∈ Dom G, y ∈ G(x)|Noncnt(y) = 0)
                if x ∈ Dom c3 then
                    c3(x) := c3(x) + {y}; else
                    c3(x) := {y};
                endif;
            end ∀;
```

To handle c_7, which is invariant within the *while* loop, we only need to define it at the end of the *while* loop prologue. The code to do this is

(17)
```
/* USER supplies the name Rhstont for c7 */
Rhstont := nullset;
(∀x ∈ Dom G, t ∈ G(x))
     if t ∈ Dom Rhstont then
          Rhstont(t) := Rhstont(t)+{x}; else
          Rhstont(t) := {x};
     endif;
end ∀;
```

The current stage of reduction ends after we replace the setformer {w ∈ *Dom* G | t ∈ G(w)} occurring in (15) by the map retrieval operation Rhstont(t).

Proceeding now to the next outer expression c_4 which contains c_3, we note that the final form of the prederivative of c_4 relative to the change to c_3 occurring in (15) is

(18)
```
c4(q) := c4(q) + 1;
```

Since (18) contains no uses of c_3 we can replace the assignment to $c_3(q)$ in (15) by the assignment (18). After this we can replace (16) by the following code which initializes c_4,

(19)
```
c4 := nullset;
(∀x ∈ Dom G, y ∈ G(x) | Noncnt(y) = 0)
     if x ∈ Dom c4 then
          c4(x) := c4(x) + 1; else
          c4(x) := 1;
     endif;
end ∀;
```

This makes the workset $c_5 = \{x \in Dom\ G | x \notin s\ \&\ c_4(x) \neq 0\}$ ready for reduction. The prederivative of c_5 relative to the change (18) in c_4 is

(20)
```
if c4(q) = 0 & q ∈ Dom G & q ∉ s then
    c5 := c5 + {q};
endif;
```

while the prederivative of c_5 relative to the strict element addition s := s + {n} is

(21)
```
(∀y ∈ {n} | y ∈ Dom G & c4(y) ≠ 0)
    c5 := c5 - {y};
end ∀;
```

which can be easily simplified to

(21')
```
if c4(n) ≠ 0 then
    c5 := c5 - {n};
endif;
```

Since we cannot eliminate c_4 , we will refer to it by a new user supplied name, Ntcnt. c_5 which the user has named W can be made available on entry to the *while* loop by executing w := {x ∈ *Dom* G|x ∉ s & c_4(x) ≠ 0};

The result of this last series of transformations is the following efficient low level version of (6),

```
(22) 1      s := {n ∈ Dom G | A ∈ G(n)};
             /* prologue */
     2       (∀x ∈ Dom G, y ∈ G(x))
     3            Noncnt(y) := [+: z ∈ y | z ∉ s]1;
     4       end ∀;
     5       Ntinrhs := nullset;
     6       (∀x ∈ Dom Noncnt, y ∈ x)
     7            if y ∈ Dom Ntinrhs then
     8                 Ntinrhs(y) := Ntinrhs(y)+{x}; else
     9                 Ntinrhs(y) := {x};
     10           endif;
     11      end ∀;
     12      Ntcnt := nullset;
     13      (∀x ∈ Dom G, y ∈ G(x) | Noncnt(y) = 0)
     14           if x ∈ Dom Ntcnt then
     15                Ntcnt(x) := Ntcnt(x) + 1; else
     16                Ntcnt(x) := 1;
     17           endif;
     18      end ∀;
     19      Rhstcnt := nullset;
     20      (∀x ∈ Dom G, t ∈ G(x))
     21           if t ∈ Dom Rhstcnt then
     22                Rhstcnt(t) := Rhstcnt(t) +{ x}; else
     23                Rhstcnt(t) := { x};
     24           endif;
     25      end ∀;
     26      W := {x ∈ Dom G|x ∉ s & Ntcnt(x) ≠ 0};
```

```
        /* main loop */
27      (while ∃ n ∈ W)
28           (∀t ∈ Ntinrhs(n))
29                if Noncnt(t) = 1 then
30                     (∀q ∈ Rhstcnt(t))
31                          if Ntcnt(q) = 0 & q ∉ s then
32                               W := W + {q};
33                          endif;
34                          Ntcnt(q) := Ntcnt(q) + 1;
35                     end ∀;
36                endif;
37                Noncnt(t) := Noncnt(t) - 1;
38           end ∀;
39           if Ntcnt(n) ≠ 0 then
40              W := W - {n};
41           endif;
42           s := s + {n};
43      end while;
```

Observe that (22) represents an improvement over (6) only when the number m of nonterminals (in G) from which the empty string can be derived is large. If we let

$$\ell = \sum_{n \in Dom\ G} \sum_{t \in G(n)} \#\ t$$

then the expected cost of executing (6) is proportional to $\ell \times m$, while the expected cost of (22) is $O(\ell)$. In analyzing (22), we see that the preprocessing cost is much

more expensive than the *while* loop. Indeed, ℓ elementary steps are consumed by the code from lines 2 to 10 of (22) where the maps Noncmt and Ntinrhs are computed. The time cost of the remaining code within the prologue is bounded by #G. However, the expected cost of the *while* loop is only proportional to $\sum_{n \in s}$ #Ntinrhs(n) where s is the set of nonterminals in G and Ntinrhs(n) is the set of all right-hand side terms which contain the nonterminal symbol n.

In comparing the preceding example with the bubble sort example, we cannot help but note that despite the potential improvement in transformational mechanization suggested by our derivation of the grammar algorithm, neither example shows any overwhelming speedup. The next two examples will exhibit much greater degrees of speedup.

In Chapter 4 (D) we obtained an order of magnitude speedup by formally differentiating a restricted version of Haberman's Banker's Algorithm. In the following example, we apply FD to an unrestricted version of the Banker's Algorithm and realize a logarithmic speedup in general, and an order of magnitude improvement if the cost of preprocessing can be neglected.

The general Banker's Algorithm considers a bank with several kinds of currency. For each kind i of currency R, cash(i) represents the total amount of this currency controlled by the bank; loan(i,c) is the loan of type i currency presently out to customer c; claim(i,c) is a

customer's claim for type i currency. The strategy of the general algorithm is the same as that of the simplified version presented in Chapter 4; i.e., any customer whose full claim can be met by the bank can be satisfied. But since we now have R different currencies, the bank can only meet the demands of a customer c if the predicate $\forall i \in R | claim(i,c) \leq cash(i)$ holds.

A base form version of the full Banker's Algorithm can be written as follows:

```
(23) 1      (while ∃ c ∈ Cus | (∀i ∈ R | claim(i,c) ≤ cash(i))
     2           (∀i ∈ R)
     3                cash(i) := cash(i) + loan(i,c);
     4           end ∀;
     5           cus := cus - {c};
     6      end while;
```

This executes in time proportional to $(\#cus)^2 \times \#R$ on the average. In order to differentiate (23) we must first transform it into a more convenient form. This can be done by applying the following sequence of transformations to the *while* loop predicate: P3 and P13 of Appendix D (V), R6, R4, and R5 of Appendix D (ii), and finally P6 of Appendix D (v). The code which results is

```
(24) 1      (while ∃c ∈ {u ∈ Cus|[+: y∈{i∈R|claim(i,u) >
                                        cash(i)}]1 = 0})
     2          (∀i ∈ R)
     3              cash(i) := cash(i) + loan(i,c);
     4          end ∀;
     5          cus := cus - {c};
     6      end while;
```

This form of the Banker's Algorithm is especially amenable to FD, since only one user directive,

```
(25)        $FD,1,Gcus = {u∈Cus|[+: y ∈ {i ∈ R |
                        claim(i,u) > cash(i)}]1 = 0},
```

is needed to speed up (24). The program analysis applied to process the command (25) will recognize that the expressions $c_1(u) = \{i \in R | claim(i,u) > cash(i)\}$, $c_2(u) = [+: y \in c_1(u)]1$, and $c_3 = \{u \in cus | c_2(u) = 0\}$ are all reducible. The reduction procedure will handle c_3 by first differentiating the innermost subexpression c_1 of c_3. The prederivative code for c_1 with respect to the change to cash(i) at line 3 of (24) is

```
(26) (while xmin(i) < cash(i) + loan(i,c))
          (∀x ∈ {w ∈ Dom claim{i}|claim(i,w) = xmin(i)})
               c1(x) := c1(x) - {i};     /* strict deletion */
          end ∀;
          xmin(i) := succ(i,xmin(i));
     end while;
```

The auxiliary maps xmin and succ occurring in (26) are initialized by executing the following code,

```
(27)    c1 := nullset;
        (∀x ∈ R, t ∈ Dom claim{x}|claim(x,t) > cash(x))
            if t ∈ Dom c1 then
                c1(t) := c1(t) + {x}; else
                c1(t) := {x};
            endif
        end ∀;
        (∀x ∈ R)            /* sort claim{x} and produce succ(x)*/
            sortas(Dom claim{x}, x)
            xmin(x) := [min: y∈Dom claim{x} |
                        claim(x,y) > cash(x)]claim (x,y);
        end ∀;
```

just prior to line (1) of (24).

However, for (26) to be profitable, we must reduce the costly set former $c_4(i) = \{w \in Dom\ claim\{i\} | claim(i,w) = xmin(i)\}$. c_4 depends discontinuously on xmin(i) and on i. Unfortunately, a general expression of the form $c(q_1,q_2,q_3) = \{w \in F_1(q_1) | F_2(q_2,w) = q_3\}$ to which c_4 can be matched will not usually be profitably reducible. The cost of initializing c is exorbitant; this can be observed by inspection of the following initialization code,

```
(28)      c := nullset;
          (∀t1 ∈ Dom F1, t2 ∈ Dom F2, w ∈ F1(t1))
               if [t1,t2,F2(t2,w)] ∈ Project(c,3) then
                    c(t1,t2,F2(t2,w))
                         := c(t1,t2,F2(t2,w)) + {w}; else
                    c(t1,t2,F2(t2,w)) := {w};
               endif;
          end ∀;
```

This code executes in time proportional to # (*Dom* F_1) × # (*Dom* F_2) × n, where n is a uniform bound on $\#F_1(t_1)$. However, when the discontinuity parameters q_1 and q_2 in c can both be replaced by 2 occurrences of a single discontinuity parameter q, (28) can be improved to the following more efficient code,

```
(29)      c := nullset;
          (∀t ∈ Dom F1, w ∈ F1(t1) | t ∈ Dom F2)
               if [t,F2(t,w)] ∈ Project(c,2) then
                    c(t,F2(t,w)) := c(t,F2(t,w))+{w}; else
                    c(t,F2(t,w)) := {w};
               endif;
          end ∀;
```

Note that (29) is formed from (28) by eliminating the second component of c, and also by turning the iterator $t_2 \in$ *Dom* F_2 within (28) into a membership test in (29).

This technique can be generalized to a rule for efficiently handling expressions which depend on redundant discontinuity parameters. Such a rule, if incorporated into the implementation design of Chapter 4, will significantly expand the contexts in which FD can be profitably applied.

In the case of c_4 , xmin(i) and i are the only parameters on which c_4 depends and which undergo modifications within the *while* loop beginning at line 1 of (24). Thus, to reduce c_4 we only need to initialize it at loop entry. The code to do this is based on (29) and is,

```
(30)      c4 := nullset;
          (∀t ∈ Dom claim, w ∈ Dom claim{t} | t ∈ Dom claim)
               if [t, claim(t,w)] ∈ Project(c4,2) then
                    c4(t,claim(t,w)) :=
                         c4(t,claim(t,w)) + {w}; else
                    c4(t, claim(t,w)) := {w};
               endif;
          end ∀;
```

where the membership test t ∈ *DOM* claim appearing in line 2 above can be removed as superfluous.

The next step in reducing Gcus requires reduction of c_2. The prederivative of c_2 relative to the change $c_1(x) := c_1(x) - \{i\}$ within (26) is

```
(31)      c2(x) := c2(x) - [+: v ∈ {i}]1;
```

which simplifies to

(31') $c_2(x) := c_2(x) - 1;$

Since (31') shows that c_2 does not depend on c_1, all definitions to c_1 in the program are dead. Thus, we can replace the change to c_1 in (26) by (31') and the initializing code (27) by the following code,

```
(32)    c2 := nullset;
        (∀x ∈ R, t ∈ Dom claim{x} | claim(x,t) > cash(x))
            if t ∈ Dom c2 then
                c2(t) := c2(t) + 1; else
                c2(t) := 1;
            endif;
        end ∀;
        (∀x ∈ R)
            sortas(Dom claim{x}, x)  /* sort claim{x} */
            xmin(x) := [min: y ∈ Dom claim{x}|y > cash(x)]y;
        end ∀;
```

Finally c_3 must be reduced. The prederivative of c_3 relative to the change (31') is

```
(33)    if x ∈ Cus & c2(x) = 1 then
            c3 := c3 + {x};
        endif
```

where the membership test x ∈ Cus within (33) can be eliminated. The prederivative code for c_3 relative to the element deletion Cus := Cus - {c} is

```
(34)      (∀y ∈ {c} | c2(y) = 0)
               c3 := c3 - {y};
          end ∀;
```

which after cleanup becomes the following equivalent code,

```
(34')          c3 := c3 - {c};
```

Since (33) contains a use of c_2 , c_2 must remain in the program under a user supplied name, say Count. A last reduction step inserts the assignment Gcus := {x∈Cus|Count(x)=0} at the end of the prologue.

The low level SETL version of the Banker's Algorithm derived from (23) is as follows:

```
(35)      /* Prologue */
     1    Count := nullset;
     2    (∀x ∈ R, t ∈ Dom claim{x}|claim(x,t) > cash(x))
     3         if t ∈ Dom Count then
     4              Count(t) := Count(t) + 1; else
     5              Count(t) := 1;
     6         endif;
     7    end ∀;
     8    (∀x ∈ R)
     9         sortas(Dom claim{x}, x);
     10        xmin(x) := [min: y ∈ Dom claim{x}|y>cash(x)]y;
     11   end ∀;
```

```
12      Ycus := nullset;  /* Ycus  replaces the name c4*/
13      (∀t ∈ Dom claim, w ∈ Dom claim{t})
14         if [t, claim(t,w)] ∈ Project(Ycus, 2) then
15            Ycus(t, claim(t,w)) :=  Ycus(t,claim(t,w))
                                      + {w}; else
16            Ycus(t,claim(t,w)) := {w};
17         endif;
18      end ∀;
19      Gcus := {x ∈ Cus | Count(x) = 0};
        /* Main loop */
20      (while ∃c ∈ Gcus)
21         (∀i ∈ R)
22            (while xmin(i) < cash(i) + loan(i,c))
23                     (∀x ∈ Ycus(i, xmin(i)))
24                        if count(x) = 1  then
25                           Gcus := Gcus + {x};
26                        endif;
27                        Count(x) := Count(x) - 1;
28                     end ∀;
29                     xmin(i) := succ(i, xmin(i));
30                  end while;
31                  cash(i) := cash(i) + Loan(i,c);
32               end ∀;
33               Gcus := Gcus - {c};
34      end while;
```

Note that the dead assignment Cus := Cus - {c} has been eliminated just after line 33 of (35).

In analyzing the expected running time of (35) we will make the following assumptions:

1. $\forall x \in R \mid \mathit{Dom}\ \text{claim}\{x\} \subseteq \text{Cus}$
2. $\mathit{Dom}\ \text{claim} \subseteq R$

We can then estimate that the cost of executing the prologue of (35) will take no more than O(#R × #Cus × log #Cus) elementary steps. This is the cost of the loop appearing in lines 8-11; the other loops within the prologue require no more time than either #R × #Cus or # Cus steps. The main loop itself should run in time proportional to #R×#Cus at most.

The preceding techniques for eliminating redundant discontinuity parameters also prove useful in the next example, a form of Kildall's iterative algorithm [KI2] for computing expressions available for a program flow graph. Input for this algorithm consists of the following,

1. the set N_F of nodes in the flow graph, where each node corresponds to a basic block of the program;
2. the set CV of potentially available expresions;
3. a map pred which maps each node $n \in N_F$ into the set pred(n) of predecessor nodes in the flow graph.
4. a preserved set map PR which maps each node $n \in N_F$ into the set PR(n) of all expressions $e \in CV$ in which there occur no definitions to parameters of e within n.

5. an exposed definitions map XE which associates each node $n \in N_F$ with the set XE(n) of all expressions e whose value is saved in a temporary σ_e by an assignment $\sigma_e := e$ occurring within n at a place after which σ_e is not spoiled in n.

The algorithm will output the set AE(n) of expressions available at the top of each node $n \in N_F$.

In SETL a base form version of the available expressions algorithm is

```
(36)        AE := nullset;
            (∀n ∈ N_F)
               AE(n) := CV;
            END ∀;
            (while ∃n ∈ N_F|AE(n)≠[*: y∈pred(n)]((AE(y)*PR(y))
                  + XE(y)))
               AE(n) := [*: y ∈ pred(n)]((AE(y)*PR(y))+XE(y));
            end while;
```

In order to improve (36) by FD, we must first take several manual steps to transform (36) into the following canonical form,

```
(37) 1      AE:= nullset;
     2      (∀n ∈ N_F)
     3         AE(n) := CV;
     4      end ∀;
     5      (while ∃n∈{m∈N_F|[+:y∈{z∈AE(m)|[+: w∈{x∈pred(m)|
                        (z ∉ AE(x) or z∉PR(x)) & z∉XE(x)}]1
               ≠ 0}]1 ≠ 0})
     6         AE(n) := AE(n)-{z∈AE(n)|[+: w∈{x∈pred(n)|
                 (z∉AE(x) or z∉PR(x)) & z∉XE(x)}]1 ≠ 0};
     7      end while;
```

Within the loop appearing in lines 5-7 of (37), there occur five different reducible expressions; these are

$c_1(m,z) = \{ x \in pred(m) \mid (z \notin AE(x)$ *or* $z \notin PR(x)) \ \&\ z \notin XE(x)\}$,

$c_2(m,z) = [+: w \in c_1(m,z)]1$, $c_3(m) = \{ z \in AE(m) \mid c_2(m,z) \neq 0\}$,

$c_4(m) = [+: y \in c_3(m)]1$, and $c_5 = \{ m \in N_F \mid c_4(m) \neq 0\}$.

(Note that c_1 is an instance of a general reducible form not currently included within our FD tables.)

The expressions c_1, c_2, and c_3 occur at both lines 5 and 6, while the other reducible expressions occur only at line 5. Unfortunately, we are unable to apply FD to (37) successfully, because the prederivative code for updating c_1, c_2, and c_3 would be executed prior to line 6, thus spoiling the old values for c_1, c_2, and c_3 which we need at line 6.

A remedy for this can be worked out by decomposing the assignment at line 6 into the following forall iterator,

```
(38)      (∀y ∈ {z ∈ AE(n) | [+: w∈{x∈pred(n) | (z∉AE(x)
                                  or z ∉ PR(x)) & z ∉ XE(x)}]1 ≠ 0})
              AE(n) := AE(n) - {y};
          end ∀;
```

After this the algorithm can be differentiated with one user directive,

```
(39) $FD,5,W = {m ∈ N_F | [+: y ∈ {z∈AE(m) | [+: w∈{x∈pred(m) |
                 (z ∉ AE(x) or z ∉ PR(x)) & z∉XE(x)}]1≠0}]1≠0}
```

In order to reduce W, its innermost reducible subexpression c_1 must be reduced first. Since c_1 contains three occurrences of the same discontinuity parameter z, we can apply the techniques of the previous case study to improve the derivative code for c_1 relative to the change AE(n) := AE- {y} occurring within (38). This involves replacement of potentially costly iterations over PR(x) and over the range CV of possible values for z by simple membership tests. In raw form before cleanup the prederivative code for c_1 is

```
(40)  (∀z ∈ {u ∈ Dom pred|n pred(u)},x∈ {y}|x∈PR(n) &x∉XE(n)})
          c1(z,x) := c1(z,x) + {n};
      end ∀;
```

where it is required that succ(n) = {u ∈ *Dom* pred|n∈pred(u)} should be reduced. To reduce succ, we have only to place the following initialization code

```
(41)  succ := nullset;
      (∀u ∈ Dom pred, n ∈ pred(u))
           if n ∈ Dom succ then
                succ(n) := succ(n) + {u}; else
                succ(n) := {u};
           endif;
      end ∀;
```

just before line 5 of (37). After straightforward application of standard cleanup transformations, (40) can

be rewritten in the following efficient form,

```
(42)  if y ∈ PR(n) & y ∉ XE(n) then
          (∀z ∈ succ(n))
               c1(z,y) := c1(z,y) + {n};
          end ∀;
      endif;
```

To complete the reduction of c_1, we place the following code within the loop prologue:

```
(43)      c1 := nullset;
          (∀m ∈ Dom pred, x ∈ pred(m), z ∈ AE(m) |
                          (z ∉ AE(x) or z∉PR(x)) & z∉XE(x)))
               if [m,z] ∈ PROJECT(c1,2) then
                    c1(m,z) := c1(m,z) + {x}; else
                    c1(m,z) := {x};
               endif;
          end ∀;
```

The next reduction step involves c_2. Its prederivative relative to the change in c_1 within (42) is simply

```
(44)          c2(z,y) := c2(z,y) + 1;
```

which makes c_1 useless within the loop. Hence, we replace the assignment to c_1 in (42) by (44), and within (43) we first differentiate c_2 relative to the changes to c_1 and then eliminate all code which refers to c_1. The code which replaces (43) is then,

(45)
```
c2 := nullset;
(∀m ∈ Dom pred, x ∈ pred(m), z ∈ AE(m)
        (z ∉ AE(x) or z ∉ PR(x)) & z ∉ XE(x)).)
    if [m,z] ∈ Project(c2,2) then
        c2(m,z) := c2(m,z) + 1; else
        c2(m,z) := 1;
    endif;
end ∀;
```

The next expression to reduce is c_3. Its prederivative relative to the change (44) to c_2 is

(46)
```
if c2(z,y) = 0 then
    c3(z) := c3(z) + {y};
endif;
```

Its prederivative relative to the change AE(n) := AE(n)- {y} is just

(47)
```
c3(n) := c3(n) - {y} .
```

Note in connection with (46) that the variable z within (46) cannot have the same value as n. This determination requires relatively simple reasoning (lending support to the possibility of proving this fact automatically), yet it is crucial to the entire reduction. After initializing c_3 by inserting the code

```
(48)    c3 := nullset;
        (∀m ∈ Dom AE, z ∈ AE(m) | c2(m,z) ≠ 0)
            if m ∈ Dom c3 then
                c3(m) := c3(m) + {z}; else
                c3(m) := {z};
            endif;
        end ∀;
```

at the end of the prologue, we can replace uses of the expression c_3 by a map retrieval opeation using the variable c_3.

All this will have put the available expression algorithm into the following transitional form,

```
(49)    1  AE := nullset;
        2  (∀n ∈ NF)
        3       AE(n) := CV;
        4  end ∀;
        5  numpred := nullset;  /* the user name for c2 */
        6  (∀m ∈ Dom pred, x ∈ pred(m), z ∈ AE(m) |
                (z ∉ AE(x) or z ∉ PR(x)) & z ∉ XE(x))
        7       if [m,z] ∈ Project(numpred,2) then
        8          numpred(m,z) := mnumpred(m,z)+1; else
        9          numpred(m,z) := 1;
       10       endif;
       11  end ∀;
       12  succ := nullset;
```

```
13  (∀u ∈ Dom pred, n ∈ pred(u))
14       if n ∈ Dom succ  then
15            succ(n) := succ(n) + {u}; else
16            succ(n) := {u};
17       endif;
18  end ∀;
19  Del := nullset;  /* Del stands for c3 */
20  (∀m ∈ Dom AE, z ∈ AE(m)|numpred(m,z) ≠ 0)
21       if m ∈ Dom Del then
22            Del(m) := Del(m) + {z}; else
23            Del(m) := {z};
24       endif;
25  end ∀;
    /* main loop */
26  (while ∃n ∈ {m ∈ NF| [+:y ∈ Del(m)]1 ≠ 0})
27       (∀y ∈ Del(n))
28            if y ∈ PR(n) & y ∉ XE(n) then
29                 (∀z ∈ succ(n))
30                      if numpred(z,y) = 0 then
31                           Del(z) := Del(z) + {y};
32                      endif;
33                      numpred(z,y) := numpred(z,y)+1;
34                 end ∀;
35            endif;
36            Del(n) := Del(n) - {y};
37            AE(n) := AE(n) - {y};
38       end ∀;
39  end while;
```

Reduction continues with handling of $c_4(m)=[+:y\in Del(m)]1$. Just prior to lines 31 and 36 where Del is modified we must insert the following prederivative code for c_4 ,

(50) $c_4(z) := c_4(z) + 1;$ /* BEFORE line 31 */

and

(50') $c_4(n) := c_4(n) - 1;$ /* BEFORE line 36 */

c_4 may also be initialized incrementally within the initializing code for Del at lines 19-25. That is, just prior to the assignment Del := *nullset* at line 19 we place the assignment c_4 := *nullset*; Then just before lines 22 and 23 we insert the incremental code

(51) $c_4(m) := c_4(m) + 1;$

and

(51') $c_4(m) :=1;$

respectively.

This brings the reduction process into a final stage in which the workset $W = \{m \in N_F | c_4(m) \neq 0\}$ can be reduced. W depends only on the definitions (50) and (50') which occur within the optimization loop. The prederivative code for W relative to these definitions are

(52)
```
if c4(z) = 0 then
     W := W + {z};
endif;
```

with respect to (50) and

(52')
```
if c4(n) = 1 then
     W := W - {n};
endif
```

with respect to (50'). It is clear from both (52) and (52') that c_4 must be included in our final algorithm. Thus, we will supply a name numdif to replace c_4. The last task to perform inserts the assignment $W := \{m \in N_F | numdif(m) \neq 0\}$ at the end of the prologue.

Collecting the results of all of the previous transformations together we arrive at the following low level SETL version of (36).

```
(53) 1    AE := nullset;
     2    (∀n ∈ N_F)
     3         AE(n) := CV;
     4    end ∀;
     /    /* prologue */
     5    numpred := nullset;
     6    (∀m ∈ DOM pred, x ∈ pred(m), z ∈ AE(m) |
             (z ∉ AE(x) or z ∉ PR(x)) & z ∉ XE(x))
     7         if [m,z] ∈ Project(numpred,2) then
     8              numpred(m,z) := numpred(m,z)+1; else
     9              numpred(m,z) := 1;
    10         endif;
    11    end ∀;
    12    succ := nullset;
    13    (∀u ∈ Dom pred, n ∈ pred(u))
    14         if n ∈ Dom succ then
    15              succ(n) := succ(n) + {u}; else
    16              succ(n) := {u};
    17         endif;
    18    end ∀;
    19    numdif := nullset;
    20    Del := nullset;
    21    (∀m ∈ Dom AE, z ∈ AE(n)|numpred(m,z) ≠ 0)
    22         if m ∈ Dom Del then
    23              numdif(m) := numdif(m) + 1;
```

```
24              Del(m) := Del(m) + {z}; else
25              numdif(m) := 1;
26              Del(m) := {z};
27          endif;
28      end ∀;
29      W := {m ∈ N_F | numdif(m) ≠ 0};
        /* main loop */
30      (while  n ∈ W)
31          (∀y ∈ Del(n))
32              if y ∈ PR(n) & y ∉ XE(n) then
33                  (∀z ∈ succ(n))
34                      if numpred(z,y) = 0 then
35                          if numdif(z) = 0 then
36                              W:= W + {z};
37                          endif;
38                          numdif(z) := numdif(z)+1;
39                          Del(z) := Del(z)+ {y};
40                      endif;
41                      numpred(z,y) := numpred(z,y)+1;
42                  end ∀;
43              endif;
44              if numdif(n) = 1
45                  W := W - {n};
46              endif;
47              numdif(n) := numdif(n) - 1;
48              Del(n) := Del(n) - {y};
49              AE(n) := AE(n) - {y};
50          end ∀y;
51      end while;
```

It is easily seen that our final version of the available expression algorithm executes in fewer than O(#CV × #SUCC) elementary steps, a considerable savings over the base form algorithm, (36).

BIBLIOGRAPHY

[A1] Allen, Frances F., Cocke, John, and Kennedy, Ken, "Reduction of Operator Strength," Rice University, Tech. Rep. 476-093-6, August 1974.

[AHU1] Aho, Hopcroft, Ullman, *Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[AU1] Aho and Ullmann, *Principles of Compiler Design*, Addison Wesley, 1978.

[BA] Bauer, F., *et al.*, "Towards a Wide Spectrum Language to Support Program Specification and Program Development," *Sigplan Notices*, Vol. 13, No. 12, 1978.

[B1] Balzer, Robert, Goldman, Neil, and Wile, David, "On the Transformational Implementation Approach to Programming," UCS/Information Sciences Institute, Marina del Rey, Calif., April 1975.

[B2] Burstall, R. M., and Darlington, J., "A Transformation System for Developing Recursive Programs," *JACM*, Vol. 24, 1 (Jan. 1977).

[BU] Burge, William, "An Optimizing Technique for High Level Programming Languages," IBM Research Tech. Report, 1976.

[C1] Cocke, John and Kennedy, Ken, "An Algorithm for Reduction of Operator Strength," *CACM* Vol. 20, 11 (Nov. 1977).

[C2] Cocke, John and Schwartz, J. T., *Programming Languages and Their Compilers*, Courant Institute of Mathematical Sciences, New York University, 1969.

[D] Deak, Edith, *Thesis* in progress, Department of Computer Science, New York University, 1978.

[DGS1] Dewar, Robert, *et al.*, "SETL Data Structures," *SETL Newsletter* # 189.

[DGS2] *IBID, The SETL Programming Language*, Manuscript, 1978.

[E1] Earley, Jay, "High Level Iterators and a Method for Automatically Designing Data Structure Representation," Dept. of Elec. Engr. & Computer Sci., and the Electronic Research Lab.,Univ. of California, Berkeley, Calif., February 1974.

[E2] Earley, Jay, "High Level Operations in Automatic Programming," Dept. of Elec. Engr. & Comp. Sci. and the Electronics Res. Lab., Univ. of Cal., Berkeley, Calif., October 1973.

[F1] Fong, Amelia C. and Ullman, Jeffrey D., "Induction Variables in Very High Level Languages," *Proc. Third ACM Symp. on Principles of Programming Languages*, January 1976.

[F2] Fong, A. C., "Elimination of Common Subexpressions in Very High Level Languages," *Proc. 4th ACM Symposium on Principles of Programming Languages*, Jan. 1977.

[G1] Goldstine, H. H., *The Computer from Pascal to Von Neumann*, Princeton University Press, Princeton, New Jersey, 1972.

[H1] Hecht, Matthew, *Flow Analysis of Computer Programs*, North-Holland, 1977.

[K1] Kennedy, Ken, "Reduction in Strength Using Hashed Temporaries," *SETL Newsletter* No. 102, March 12, 1973.

[K2] Kennedy, Ken, "Linear Function Test Replacement," *SETL Newsletter* No. 107, May 20, 1973.

[K3] Kennedy, Ken, "Global Dead Computation Elimination," *SETL Newsletter* No. 111, August 7, 1973.

[K4] Kennedy, Ken, "An Algorithm to Compute Compacted Use Definition Chains," *SETL Newsletter* No. 112, August 14, 1973.

[K5] Kennedy, Ken, "Variable Subsumption with Constant Folding," *SETL Newsletter* No. 123, February 1, 1974.

[KI1] Kibler, D. F., *et al.*, "Program Manipulation via an Efficient Production System," *SIGPLAN*, Aug. 1977.

[KI2] Kildall, Gary A., "A Unified Approach to Global Program Optimization," *Proc. First ACM Symp. on Principles of Programming Languages*, Oct. 1973.

[L2] Loveman, D. B., "Program Improvement by Source to Source Transformation," *JACM*, Vol. 24, 1 (Jan. 1977).

[P1] Pratt, Terrence, *Programming Languages: Design and Implementation*, Prentice-Hall, 1975.

[P2] Paige, Bob, and Schwartz, J. T., "Expression Continuity and the Formal Differentiation of Algorithms," *Proc. Fourth ACM Symp. on Principles of Programming Languages*, Jan. 1977.

[S1] Schonberg, Ed, "The VERS2 Language of J. Earley Considered in Relation to SETL," *SETL Newsletter* No. 124, Jan. 30, 1974.

[Sch1] Schwartz, J. T., *On Programming: An Interim Report on the SETL Project*, Installments I & II, (one vol.) Courant Inst. Math. Sci., New York Univ., New York, 1974.

[Sch2] Schwartz, J. T., "General Comments on High Level Dictions, and Specific Suggestions Concerning 'Converge' Iterators and Some Related Dictions," *SETL Newsletter* No. 133-B, Jan. 29, 1975.

[Sch3] Schwartz, J. T., "Introductory Lecture at the June June 28 'Informal Optimization Symposium'," *SETL Newsletter* # 135, July 1, 1974.

[Sch4] Schwartz, J. T., "Structureless Programming, or the Notion of 'Rubble', and the Reduction of Programs to Rubble," *SETL Newsletter* # 135-A, July 12, 1974.

[Sch5] Schwartz, J.T., "A Framework for Certain Kinds of High Level Optimization," *SETL Newsletter* No. 136, July 16, 1974.

[Sch6] Schwartz, J. T., and Paige, B., "On Jay Earley's 'Method of Iterator Inversion'," *SETL Newsletter* No. 138, April 19, 1976.

[Sch7] Private communication.

[Sch8] *Ibid* "Optimization of Very High Level Languages," Parts I,II, *J. of Computer Languages*, pp. 161-218, 1975.

[Sch9] *Ibid.*, "Updating the Lower Bound of a Set of Integers in Set Theoretic Strength Reduction," *SETL Newsletter* No. 138-B, 1976.

[Sch10] *Ibid.*, "On the 'Base Form' of Algorithms," *SETL Newsletter* # 159, Nov. 1975.

[Sch11] *Ibid.*, "A Higher Level Control Diction," *SETL Newsletter* # 133, June 1974.

[Sch12] *Ibid.*, "Additional Pursue Block Examples," *SETL Newsletter* # 133-A, July 1974.

[St1] Standish, Thomas, "An Example of Program Improvement Using Source-to-Source Transformations," Dept. of Information and Computer Science, Univ. of Cal. at Irvine, Irvine, Cal., February 11, 1976.

[St2] Standish, Thomas, et al., "The Irvine Program Transformation Catalogue," Dept. of Information and Computer Science, Univ. of Cal. at Irvine, Jan. 7, 1976.

[T1] Tenenbaum, A., *Thesis*, New York Univ., Oct. 1974.

[W1] Warren, Henry, *Thesis*, in preparation, Department of Computer Sci., New York Univ., 1978.

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012