ON THE SELECTION AND IMPLEMENTATION OF DATA STRUCTURE REPRESENTATIONS

R. B. GIMSON

Ph.D. Thesis March 1978

University of Newcastle upon Tyne

ACKNOWLEDGEMENTS

I would like to thank Dr. Peter Henderson for his supervision during the research, for his patience during its writing up and for his helpful comments on its presentation.

Thanks also to Mrs. Jill Kerr for her rapid and accurate typing of the thesis, and to past and present colleagues at Newcastle for providing the right environment for the work to be carried out.

The research was supported by the Science Research Council.

ABSTRACT

The selection and the implementation of representations for the data used in a computer program are considered in order to see what assistance can be provided to the programmer in carrying out these responsibilities. The notational and system support required to enable a library of generally applicable data representations to be established and used during program development are investigated.

An approach to data representation has been developed which is based on transformations applied to the source language form of a program. A description is given of a notation for expressing such transformations in a form suitable for inclusion in a library of representations. An experimental system to aid selection and implementation of data representations has been developed in order to investigate the consequences of adopting a transformational approach to data representation. Examples are presented of the operation of the experimental system to demonstrate how a programmer may guide the choice of representations for use in a program.

Some conclusions are drawn concerning the feasibility of the transformational approach, and its possible further development.

CONTENTS

CHAPTER 1

INTRODUCTION

1.0 Summary

This chapter describes the context in which the subsequent investigations are set. The representation of data structures is viewed as an important aspect of the programming process in which it appears feasible and useful to augment the knowledge and skill of the programmer with machine aid. The form that such aid might take is outlined, and the approach investigated in this thesis is introduced. The relationship to other work in similar areas is described.

1.1 Complexity and Programming

Over the past few years there has been a growing realisation of the difficulties that may, and often do, arise when writing computer programs. Much attention is now being given to investigating how these difficulties may be reduced, both by the use of various methodologies and by providing computer assistance with the programming task.

1.1.1 The Human Factor

Programming involves problem solving, and despite attempts to automate parts of the task, the human programmer still plays the central role in developing a program. Dijkstra [13] has pointed out that developing a successful program depends on a full appreciation of the difficulties involved, and that the intrinsic limitations of the human programmer must be respected. Weinberg [55] has considered the many ways in which human factors affect programming, and advocates a deeper study of such factors. For example, the problem solving ability of the programmer is central to the programming task, and a suitable environment is needed in which such problem solving can take place for it to be successful. The information capacity of the human mind is strictly limited in its ability to recall quickly large amounts of information. If the difficulties of programming are to be reduced, assistance (both methodological and mechanical) is required for those aspects of the task in which the programmer's ability is limited. However, at the same time we do not wish to hinder the programmer's control over the eventual form of the program. We wish to augment the capabilities of the programmer, not to replace him or her.

1.1.2 The Programming Task

Many complexities arise in programming. Constraints on the size and efficiency of the final program mean that it must often be expressed in machine oriented terms, which are very different from the machine-independent terms in which the problem to be solved is expressed. The amount of program code that has to be produced can lead to the inability of the programmer to comprehend all the code and its complex inter-relationships, which in turn adversely affects its correctness and maintainability.

The programming task can be viewed as one in which the gap between the problem to be solved and the machine on which it is to be solved must be bridged. Having established the problem concepts and worked out an overall method of solution, it is the programmer's job to produce a machine-acceptable, suitably efficient program to carry out the solution algorithm.

In order to bridge the problem-machine gap, the programmer has to choose representations for the processing and data concepts of his abstract solution and express them in terms of the basic concepts provided by the machine or language actually used to implement the solution.

The programmer is therefore faced with the difficulty of choosing suitable representations for the processing and data concepts he wishes to use in his solution. His choice is constrained by the limits on the size and execution speed of the final program, as well as by the basic programming concepts available in the language in which that program is to be written. Having chosen a representation, the programmer must also correctly implement his choice in terms of the basic concepts of the language.

1.2 The Representation of Data in Programs

For this work, we consider in particular the selection and implementation of representations for the data used in programs. The choice of which data representations should be used is a basic factor in the design of a program, but one which is far from simple. We wish to consider the effect of making it the central theme of the development of a program, so that the attention of the programmer is concentrated on making representational choices. By this means we hope to make the wide range of possible choices more explicit to the programmer, and give him better facilities for exploring that range in order to select suitable representations for his program.

Conventionally, the programmer only has his experience and intuition to help him choose a suitable representation for his abstract data concepts. The multiplicity of different ways of representing data makes the choice of representation for any specific data construct difficult. Usually data is organised in compound structures, each component of which could itself be compound. Any single component of a compound data structure could involve a representational choice from many conventional storage techniques, such as lists, trees or hash tables, as well as many unconventional storage techniques that are invented in an ad hoc fashion by programmers. So choosing representations for compound data structures can become very complex.

The programmer, faced with the need to make many representational choices, and usually under pressure to design and write a program quickly, will often not consider a lot of possible representations. He is liable to consider making a choice only from among those representations with which he is familiar, and will not have time to search the available literature for a representation more suitable for his data. In addition, his evaluation of the representations that he does consider is likely to be sketchy or even non-existant. He will possibly just pick what he intuitively feels to be the most appropriate representation, implement it in his program, and only reluctantly consider changing it when the program is found to be too inefficient.

There is therefore strong motivation for investigating whether assistance can be provided to help the programmer in selecting and implementing data representations.

1.2.1 Two Example Problems

To illustrate the kind of problem to be faced when choosing data representations, we shall introduce two examples from those used during the experimentation, one fairly simple and the other somewhat more complex. These examples will be used throughout the text to demonstrate the concepts being discussed. Here we present the problems with some initial contemplation of the variety of data representations that might be used in their solution.

Birthdays Example (Initial problem): An unordered file of data is given containing the day in the year on which each of a collection of people were born. Each entry in the file consists of a pair of integers, the first representing the identification number (in the range 1 to 5000) of a person and the second representing their birthday (in the range 1 to 366). There are about 50, and not more than 100 pairs of data on the input file. It is required to list, for each day of the year for which there is at least one birthday, the identification number of each person having a birthday on that day.

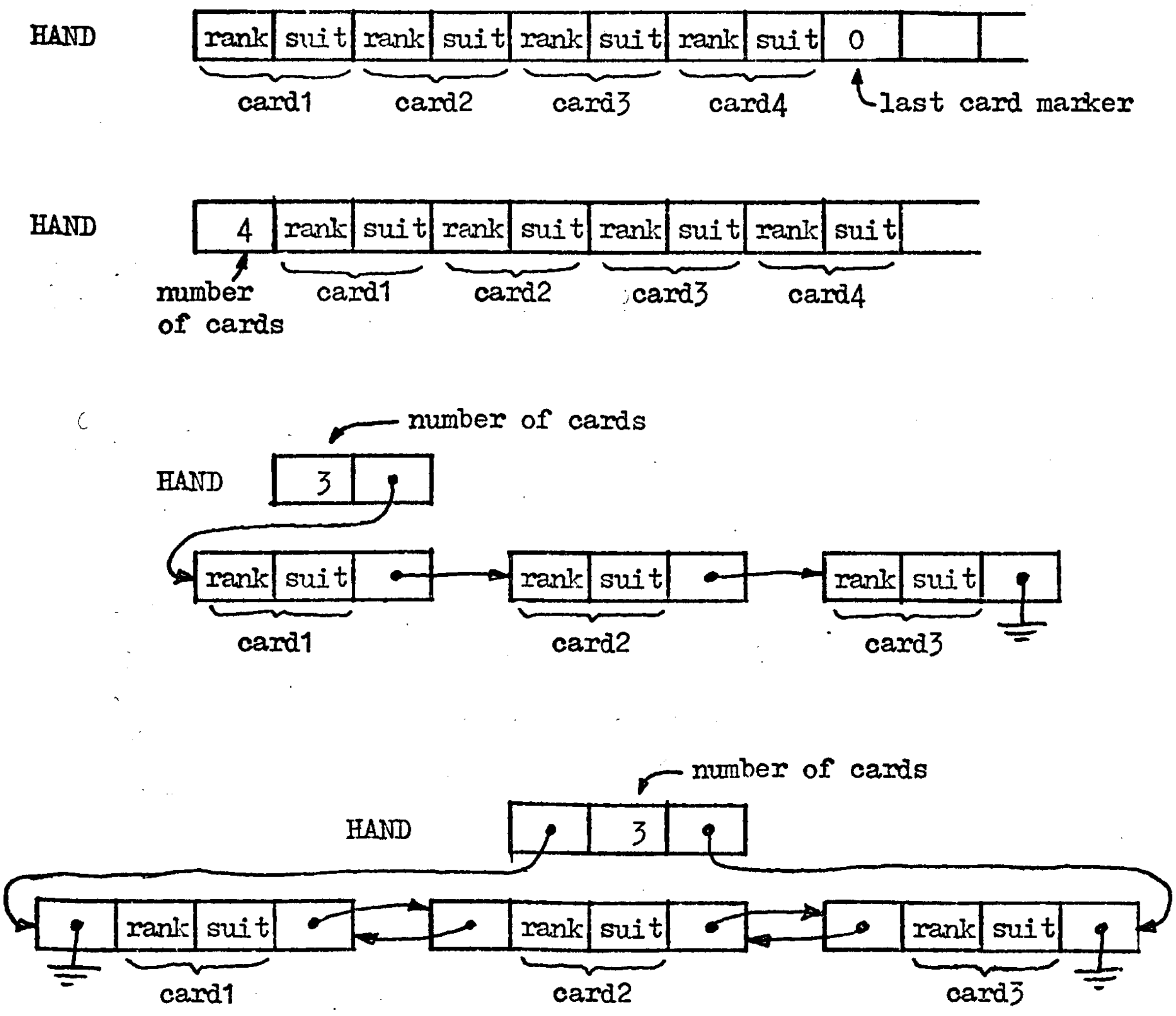
In this example, we require a data representation for the storage of the data during the execution of the solution. It is necessary to store the input data in the program since it is not possible to determine the first item to be output until the last item has been input. The data may be stored in many ways. It is necessary to choose a storage representation which is practicable and suitably efficient for a proposed implementing machine, in terms of both storage capacity and execution time.

Card Game Example (Initial problem): It is required to model the playing of a card game. In this game there are two players who use a standard 52-card deck of playing cards. The game proceeds as follows. The cards are shuffled and placed in a pile face down. Each player takes seven cards from the pile. The next card on the top of the face down pile is turned over to form the bottom card of a new face-up pile. The players then take turns with the object of discarding all the cards in their hand, the winner being the first player to achieve this. At each turn a player may place one card from his hand face up on the top of the face-up pile but only if that card is either the same suit or the same rank as the card currently top of the face-up pile. If no discard can be made in this way, the player must pick up the top card of the face-down pile.

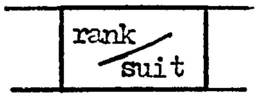
This example is chosen to demonstrate a case in which several data structures are closely related in one problem. Representations must be chosen for the playing cards and the way in which they form two piles, one face-up and the other face-down, each used in a different way. The hands of the two players, and the insertion of cards into and removal of cards from the hands, must also be represented. How might a programmer represent the data involved, say in denoting the hand of one of the players ? Possibly he will draw some schematic diagrams of how the available storage cells will be used, as shown in Figure 1.1. However, even having drawn such diagrams, the programmer is then still left with the problems of whether a better representation has been overlooked, how to evaluate and choose between the different representations, and how to code the pieces of program that are required to implement the chosen representation. Clearly a more systematic approach to choosing data representations could be of great help.

1.3 Assisting the Programming Task

Many different approaches are being tried by many people in an attempt to assist the programming task. In recent years the whole field has assumed more importance; the complexity of the problems being tackled has increased, which has in turn shown the inadequacies in the previous ad hoc approaches to programming. Attempts are being made to provide assistance both during the development of a program and in guiding



In any of the above, the rank and suit could be packed into one cell:



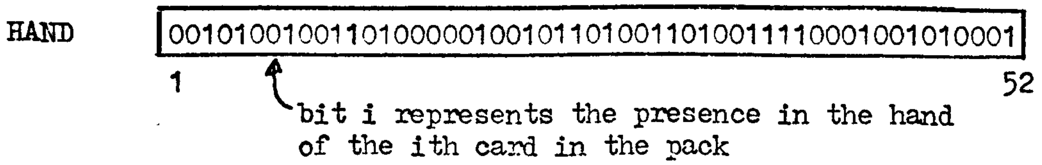

HAND
001010010011010000001001011010011010011110001001010001
1
52
bit i represents the presence in the hand of the ith card in the pack

Figure 1.1 Diagrams showing some representations for a hand of cards

the form that the completed program should take. Programming methodologies seek to provide a framework which the programmer can adopt in his approach to the task, and computer based tools can give mechanical aid to the various stages of producing a program. In addition, programming languages and notations have a marked, if not disproportionate, influence on the programming task, and much work has been carried out on devising new languages that try to provide features more closely related to a given problem domain.

1.3.1 Some Approaches

Programming languages, as pointed out by Cheatham [7], have evolved a great deal towards including facilities to aid the programmer in coding his solution. Language concepts such as the procedure or subroutine allow the programmer to write a program in manageable sections. High-level languages such as Algol 60[44] include control structures that can be understood more easily than the labels and jumps of assembler code. Extensible languages [19,57] enable the programmer to a certain extent to include data and operations more suited to his problem. Very-high-level languages, such as SETL [47] or SAIL [18], include general set-theoretic concepts, so that the programmer can express his problem solution in an almost completely machine-independent fashion.

Besides programming methodologies that advocate various kinds of stepwise approach to program development, the programmer may be assisted by computer-based tools that support each development. Experiences, such as that of the TOPD system[23,24], have shown how database support for the growing program during development enable the programmer to concentrate on elaborating the abstract concepts in his solution in a controlled manner.

One of the other ways in which the amount of program code that has to be written can be reduced is by using existing routines to carry out the more straightforward calculations. In some application areas, for example numerical analysis and statistics, libraries of useful programs and routines have been established which the programmer can link into his own program. Such re-use of existing tried components both reduces the workload on the programmer and may increase confidence in the correctness of the components.

Another approach to easing the programming task is that of the program manipulation system, as described by, for example, Knuth ([32] pg282). Using such a system, the programmer first specifies his program in an easily expressible but possibly inefficient form, then the program is transformed by the system into an acceptably efficient one. Such a system may be completely automatic, for example a conventional optimiser as included in some compilers, or may be used interactively by the programmer to help guide the choice of the transformations to be applied.

1.3.2 Assisting Data Representation

It is natural to consider how these sorts of approaches may be applied to helping the choice of data representations in programs. The programmer must be aware of the range of representations available for him to use, a range which may increase as new representations are devised. Following the idea of the use of libraries of useful numerical and other algorithms, it appears beneficial to establish, if possible, a library of useful data representations, which could be extended as suitable new representations were found.

In order to make effective use of such a data representation library, techniques are required for selecting from the library, evaluating suitable representations for use in programs, and implementing the chosen representations in the programs.

Other approaches have been attempted to help the programmer make effective use of data in programs. We suggest that some of these approaches have not gone far enough, and some of them have gone too far in the amount of assistance given to the programmer. For example, the class notation of Simula 67 [10] is a language feature which provides a good way of expressing problem-oriented data concepts required in the development of a program. The body of the Simula class can be used to define the representation of the class of data objects in question, in terms of both the storage for and operations allowed on those objects. The class construct is therefore suitable for expressing the representation of a particular class of object in a particular program. However it is not adequate to allow a library of different representations for each of a set of general data concepts to be set up (as will be seen later) nor does it ease the difficulties of selecting appropriate representations. In this sense, the Simula class does not go far enough in helping the choice of data representation.

A second approach to assisting the programmer is that of the so called very-high-level language such as SETL. In this language, the main way of expressing the problem is in set-theoretic terms. The language itself provides an implementation of general sets, so that the programmer does not need to get involved in representing his problem-oriented concepts at all. However, this is going too far in 'assisting' the programmer since the given implementation of sets will often be inappropriate and therefore inefficient for the particular use made of the sets by the programmer.

More recently, the designers of SETL have turned to investigating ways in which more control can be introduced over the representations used in implementing SETL programs (Schwartz [47,48]).

We feel it to be essential that the programmer should be able to use representations for the data which are suitably efficient for his particular problem. This means that ultimately the control over the choice of representations must remain with the programmer, and we wish to assist him make a suitable choice, rather than to impose an unsuitable choice upon him.

1.4 A Transformational Approach to Data Representation

In this thesis we present a transformational approach to the representation of data in programs, and report on the investigations carried out to examine how the approach can assist the selection and implementation of data representations.

The work is directed specifically at the representation of the data used in individual programs, where the manipulations to be carried out on the data are to a large extent known. A different approach may be required in situations, such as the representation of the data in databases, where the use made of the data is often more diverse and unpredictable.

1.4.1 Objectives

The objectives of the work were therefore to examine the problems involved in describing and setting up a library of useful data representations, expressed as transformations, also to see if such a library could be used to help the programmer in the selection and implementation of data representations for his programs.

The program transformation approach has already been applied to local program optimisation and to control flow manipulation (as discussed in the next section). In the approach, having written an initial program, successive transformations may be applied to it which, while preserving the specified action of the program on execution, alter other features of the program (such as its efficiency) in a beneficial way.

The transformational approach has also been adopted as one means of obtaining program correctness. By starting with an obviously correct program, the application of provably correct transformations enable an acceptable final program to be obtained which it would be difficult to prove correct in isolation.

Our objective is to adapt the approach so that the transformations consist of the implementations of chosen data representations. By starting with an initial program which uses abstract representation - independent data concepts, successive transformations may be applied to it in order to produce an efficient version of the program which uses only concrete data concepts which are directly supported by a programming language implementation.

In order to apply this approach, means of expressing data representations as transformations are required in a form that allows them to be included in libraries. Their selection and implementation from the libraries should be straightforward, and the range of representations that can be incorporated in programs in this way should be large and unconstrained.

Since the selection and implementation of representations is likely to involve actions (such as searching libraries and consistently applying transformations to programs) which the human programmer is not well suited to carry out, the extent to which machine aid can be applied to assisting these actions is also of concern in the investigation of the transformational approach.

1.4.2 Approach

The approach taken to the above objectives was to develop an experimental system for aiding the choice of data representations that could act as a test-bed for the associated ideas. Because of the current lack of knowledge concerned with describing data representations in a concise enough form for use with a machine, it is initially necessary to investigate the problems to which this approach may give rise. The experimental system therefore was developed to allow different ideas to be tried in order to discover the problems and the important factors involved. This thesis expands upon the problems and factors involved in assisting the choice of data representation using a transformational approach, as brought to light by the experimental system. The system was not designed as a model for a final production system, and the efficiency of the system implementation and its human engineering aspects, for example, were not therefore of prime consideration. The examples of the use of the system given later are intended to illustrate points concerned with data representational choice rather than to show how a practical system might appear.

1.4.3 Benefits to be Gained

By expressing data representations as transformations in a systematic fashion, certain benefits may be gained.

- The range of choice from which a representation for a particular data concept may be chosen becomes explicit, giving less chance that a suitable representation may be overlooked.
- The algorithms required to implement the representation are presented in a form suitable for direct application within a program, reducing programming effort by such re-usable components.

- The representation description is complete, and may be tested before inclusion in a representation library. It is also more amenable to the kind of proof correctness described by, for example, Hoare [26]. Use of such pre-tested components is an aid to program reliability.
- Describing transformations in a machine-readable form, enables machine aid to be used in their selection and implementation.
- Machine assistance in selecting representation enables larger libraries of representations to be contemplated than the unaided programmer could reasonably search.
- Machine application of the transformations when implementing a representation avoids possible programmer-introduced errors in their implementation.

1.5 Relation to Other Work

The present work is related to, and brings together, two main fields of current interest. These are, firstly, the transformational approach to program development, and secondly the provision of assistance in the selection of data representations.

1.5.1 Work on Program Transformation

In one of the first of a growing list of published work on program transformation, Knuth [32] suggests how an interactive program manipulation system could be used to enable a programmer to successively transform a correct, well-structured but possibly inefficient program into a suitably efficient form.

Various other authors have also considered the application of transformations in programming [1,3,8,49,54], mainly in the area of local optimisations to program segments. Some of these authors propose the development of interactive program manipulation systems, and give some examples of the kinds of transformation they hope might be achieved through the use of the systems, however published experience of the development of their systems is not yet available.

Transformational systems which have been developed include that of Darlington & Burstall [11,6], who provide sets of transformation applicable to programs expressed as first order recursion equations. Their transformations include recursion removal, elimination of common subexpressions, and in-line expansion of procedure bodies. Loveman [38] describes various transformations that can be applied to a tree-structured representation of a program, also mainly to achieved local optimisations of data access and control flow. A recent paper by Kibler et. al.[30] describes how an interactive program manipulation system was implemented using a special technique for organising sequences of program transformations. They give an example of the kind of optimisations that can be carried out on a matrix multiplication program.

Though some authors, including Knuth [32] and Balzer et.al.[3] suggest that the transformational approach could be applied to the representation of data as well as to the optimisation of program segments, detailed considerations of how such an approach could be carried out have not yet been made.

The correctness of transformations applied to programs is considered by Gerhart [20] .

1.5.2 Work on Assisting Data Representation.

The need for a means of isolating data representational choices has been pointed out for several years (Balzer [2], d'Imperio [15], Mealy [42]), however it is only fairly recently that any work has been carried out on providing automatic assistance to the programmer in this task.

Low [39] describes a system designed to demonstrate the feasibility of automatically selecting low-level data structures for a program expressed in terms of high-level structures based on relations, sets and sequences. His system works in three phases. Firstly, it analyses the program written in a high-level language (a subset of SAIL) which uses the data structuring methods of lists and sets. The analysis gathers information on how the program manipulates the data, and attempts to partition the data into equivalence classes, in which the members of each class will use a common representation. The second phase selects a representing structure for each equivalence class from a library of fixed structures such as variously linked lists, trees and hash tables. These structures are directly implemented in an assembly language. Selection ends when a representation has been chosen for each equivalence class. The third phase consists of preparing the user's program for compilation and execution. One of the important parts of the selection phase is the evaluation of alternative structures using space and time cost estimates. This will be discussed further in section 5.3.

Low and Rovner [40,46] go on to discuss the extension of Low's system to also handle associations (ternary relations) as one of the high-level data structuring methods, besides sets and lists. The approach is similar to that already described by Low [39].

Gotlieb and Tompa [22] describe an algorithm for selecting a storage schema for some information whose logical structure has been specified. The logical structure to be represented is described using a relational model, or what Gotlieb and Tompa describe as a 'substructure model', and the operations upon the data structure required by the user are also specified. In order to choose a storage schema (i.e. a representation) for the information structure, it is first subdivided into substructures, so that a suitable representation can be chosen for each. The representations are chosen from a known set which includes such schema as binary ring, contiguous store, hash, threaded binary tree and unary chain. An algorithm is described which initially eliminates those schema which are not applicable to a given substructure, either because they conflict with previously chosen schema, or because they do not implement operations required by the user, or because they would require more storage space than is available. From the remaining schema, an efficient one must be chosen for the substructure. The evaluation technique used by Gotlieb & Tompa to select a cost-effective schema is described more fully in Tompa[51], and will be referred to in section 5.3.

Both the representational choice schemes described by Low and Tompa consider the representation of a data structure as an essentially single-step transformation from a high-level notation to a low-level implementation. The work presented here differs in that it considers data representations as being composed of a series of transformations within a single notation. In other words, the same notation is used to express the initial abstract program as is used for the final concrete program, the change from one to the other being achieved by a series of transformations each denoting a particular representational choice.

Two additional papers have recently appeared on the selection of data representations. Kant [29] describes the design of a system which aids the selection of representations for abstract constructs in a very-high-level program description. The system acts as an evaluator of the efficiency of application of refinement rules from a separate knowledge base of such rules. Rosenschein & Katz [45] present a model of the process of choosing representations which is ultimately intended as the basis of a knowledge-based interactive system. The model uses a special program-independent language for describing the data structure requirements, and considers the ways in which representations may be combined to meet these requirements.

1.6 Organisation of Subsequent Chapters

The following chapters report on the concepts that were developed and used in the experimental representational choice system and on the conclusions that can be drawn from use of the system. Little will be said about the actual implementation of the system, since, as has been mentioned, the objective was to investigate the concepts involved rather than the provision of a production system.

The next chapter takes a general look at the role of data representation in programs, how problem-oriented data concepts can be transformed into machine-oriented terms, and the part played by abstraction in program development. Some terminology that will be used in the rest of the thesis is presented, including that of data types and data structures, and the specification of the former in terms of the latter.

Chapters 3 and 4 introduce the notation that was developed to express data representations as transformation in the experimental system. In Chapter 3 we consider the kinds of representation to be expressed, and how a transformational approach can be applied to them. We then take a closer look at transforming the structural specification of data. In Chapter 4 the forms of operation involved in the manipulation of data are considered, and a classification of these operations is introduced. The transformation of data operations is given closer attention in the remainder of this chapter.

In Chapter 5 the experimental representational choice system is introduced, and its method of use described, including the matching of representations from the library. The evaluation of representations is given particular attention here. Examples of the use of the experimental system are also presented.

An assessment of the transformational approach to data representation is made in Chapter 6, which concludes by summarising the achievements of the work.

CHAPTER 2

DATA REPRESENTATION FOR PROGRAMS

2.0 Summary

In this chapter we consider the general role of data representation in programs, demonstrating the part it plays in program development. In particular, we consider abstraction, a fundemental aid to help formulate programs, and relate it to the concept of a data type. This leads to the specification of data types in terms of data structures, and we try to clarify some of the confusion that exists between data types and data structures.

2.1 Fitting the Problem Solution to the Machine.

2.1.1 The Target Language.

The objective of computer programming is to derive and express an algorithm for the solution of a given problem in a subsequently machine executable form. The algorithm will generally be expressed as a program in some kind of programming language, ranging from machine code to a 'very-high-level' language, so that the program can subsequently be compiled and executed (or interpreted) to solve the problem. The programming language, in which the fully developed program is to be expressed, will be called the target language.

It is the programmer's job to develop a target language program that can be used to solve the given problem. Sometimes the programmer, especially if he is inexperienced or believes the problem too small to warrant more care, will immediately try to code a suitable program, keeping only in his head details of the algorithm and data representations being used. However, the 'humble programmer' [13], who realises how easily small yet vital details can be overlooked in this fashion, will try to make explicit the various assumptions made at each stage in the development of the program. Indeed for large problems, which require correspondingly large programs, it is essential to break the program development down into manageable stages.

2.1.2 The Solution Algorithm.

A program reflects a certain chosen approach to solving the given problem. It is the embodiment in the target language of an algorithm for solving the problem. The derivation of a suitable algorithm, expressed in problem-oriented rather than target-language terms, is usually the first step in reaching a solution. The expression of such an algorithm can take many forms, of varying degrees of precision. Sometimes there are standard algorithms that can be used if the problem is fairly common.

> Example: If the problem is to find the average of a set of numbers, a standard algorithm is to 'find the sum of the numbers and then divide this by the number of items in the set'. Note that the algorithm is completely independent of such target-language-oriented features as the representations of the individual numbers, or of the set, or the order of taking the numbers etc.

2.1.3 Implementing the Algorithm

Having decided upon a suitable algorithm to solve the problem, expressed in problem-oriented terms, the programmer then has the task of deriving a target language program to implement that algorithm. However, usually the concepts available in the target language are very different from those of the problem. Even if the required concepts are available in the language, that language may not provide suitably efficient implementations of them for the given problem. The programmer's job therefore is to bridge the gap between the problem concepts and the language concepts in such a way that the ultimate execution of the solution on the machine is acceptably efficient.

> Example: A useful data concept when writing a compiler is that of a symbol table (with some associated operations that can be performed upon such a table, like finding whether a given symbol is currently in the table). However, if the target language in which the compiler is eventually to be written is, for example, assembler-code, the language does not directly implement the data concept of a symbol table. It is therefore necessary to express the problem-oriented data concept in terms of concepts such as words of storage which the assembler code does support.

So, in order to attain an efficient machine solution of the original problem it is necessary not only to derive and express an algorithm for the solution of the problem, but also to implement that algorithm efficiently in machine terms.

2.1.4 Raising the level of the target language.

One approach to bridging the problem-machine gap has been the development of higher level languages that try to incorporate concepts that are more suitable for expressing algorithms in problem-oriented terms. High-level languages such as Algol, and so-called very-high-level languages such as SETL, have provided means of expressing an algorithm in terms closer to that of the problems within their scope of application and so less dependent on machine features. However, when such languages use fixed implementations for their structures and operations, as embodied in their compilers and interpreters, in general the higher the level of the language, the less efficient the resultant execution of the program may become.

> Example: A symbol table and its associated operations could be expressed in set-theoretic terms, and thus be directly expressible in a set-theoretic target language. However, such a language, since it must support complex operations such as generalised set union, may implement its sets as, for example, linked lists of elements. However this fixed implementation is probably completely inappropriate and therefore inefficient for the particular restructed set of operations actually required on the symbol table in the application being considered.

In other words, the higher level language approach to bridging the problem-machine gap generally takes away from the programmer the control over the implementations ultimately used in the machine that executes the program, and thereby reduces the efficiency of execution. What we wish to do is not to take control away from the programmer, but to assist him to make more efficient use of the control that he can exercise. We must therefore consider in more detail how the problem-machine gap can be bridged in a controlled manner.

2.2 <u>Abstraction and Representation</u>.

2.2.1 Mastering Complexity

When trying to tackle any complicated task, human limitations dictate that to avoid confusion and error the task must be broken down into manageable stages. Because of the requirements for precision and efficiency in programs, program development, especially for programs of a non-trivial size, is certainly a complicated task. It is therefore necessary to apply techniques that will allow the program to be developed in easily understood stages. If we wish to express a complex problem-oriented algorithm in machine-oriented terms, it is foolish to attempt to do this in one step, since that step would itself be too complex to be fully understood. Instead, the complexity of the task has to be overcome, and a prime means for trying to understand complex phenomena is that of abstraction.

2.2.2 Procedural Abstraction.

To demonstrate the role abstraction can play in program development, we will first consider what is perhaps its most frequent use at the present, namely the procedure. Essentially a procedure is a means of grouping some program text together and giving it a single name, which can subsequently be used to stand for the given piece of program when constructing other program text. Note that we are taking 'procedure' in its most general sense, including such similar concepts as subroutines, functions and macros, and not implying any particular implementation (such as a stack for passing parameters and holding return links). At the moment we are considering program development, in which, for example, it is often useful to introduce a procedure simply to group program text into a manageable unit. (The procedure may only be called from one place and so be more efficiently implemented by being expanded in-line in the final program.) In fact, the ability of a procedure to be used to break a program into manageable pieces is exactly why it is useful to overcome complexity in program design. It allows the program designer to ignore the detail of the procedure body, and use instead simply the procedure name, in the context of the calling program section. The design of the procedure body can then be considered in its own right, using what Dijkstra [14] calls the 'separation of concerns'.

Card Game Example: A procedure may be declared so that the identifier 'play' will stand for the actions taken by one player when it is his turn to play. The programmer may then call this procedure whenever he requires these actions to be taken in the program, without considering how the computation is to be performed. He can separately decide the way in which a player acts when it is his move, and then program the body of the procedure accordingly.

Procedural abstraction is provided as a facility in most high-level programming languages, and even most assembly languages provide some macro-definition facilities. However, though procedural abstraction has received most attention in the p ast, data abstraction is now being recognised as at least of equal importance in the development of programs.

2.2.3 Data Abstraction.

In the same way that procedural abstraction allows the implementation of a procedure to be ignored at one level, and only considered in detail at an appropriate stage in the program development, so data abstraction allows the representation of a data object to be ignored at one level, and considered seperately in detail at another level. Data objects differ somewhat from procedural objects in that firstly where a procedure may have several invocations, a data type may have instead several instantiations, and secondly a single data item may be manipulated in various ways (by assignment etc.) at various points in a program. Thus a data abstraction consists at one level of the introduction of a single identifier to stand for the type of data being considered, and also the identification of those data operations which may be applied to instances of the data type. At this level the representation of the abstract type is ignored and instances of the type may be created and manipulated in abstract terms. Only at some other appropriate stage need the representation of the type be considered in detail, together with the implementation of the operations on the type.

> Birthdays Example: In the program for this problem it is necessary to somehow store the person-date values which are input, and later to retrieve them in a certain order. A data abstraction 'table' may be created for this, on which operations such as 'insert-person-date' and 'any-person-with-the-given-date?' may be defined. The program can then be written in terms of these concepts without considering at that stage the way in which the table will be represented.

2.2.4 Representation

The separation, provided by the use of abstraction, between the the introduction of some concept and its elaboration allows the elaboration to be given more precise consideration. The representation of an abstract concept in terms of more concrete concepts both for procedural and data abstractions, determines the efficiency with which that abstract concept may be handled in the program.

By determining the use made of the abstract concept in an abstract form of the program, a better choice may be made for its representation in the more concrete forms of the program. Also, the implementation details of the concept, particularly where operations on abstract data are concerned, may be better comprehended when considered separately from their abstract use.

If appropriate constructs are allowed in the target language, the representations chosen for use in the program may be distinguishable in the final program text. For example, if the target language supports procedures, procedural abstraction is directly visible in the distinction between a procedure call and the body of the procedure being called. The representation of the procedural abstraction determines the form of the procedure body.

Often, however, the target language is not well suited to expressing in the final program text the representations that have been chosen during program development. In this case, it will depend on the documentation of the program development as to whether the representations implemented in the final program are clearly distinguishable. It is important that the representations used in a program are clearly distinguishable both for making the initial implementation of them in the program manageable and comprehensible and also for allowing future changes in the program to be carried out in a controlled fashion.

2.2.5 Program Transformation.

The program development process, which takes the solution algorithm and implements it in terms of a target language, may be considered as a series of transformations that transform the abstract problem-oriented concepts into concrete target-language-oriented concepts. If such transformations can be clearly expressed, the program development process may be described by the application of a given series of transformations to an initial abstract version of the solution algorithm.

Program transformations can be performed from one form of the program expressed in one notation to another form of the program expressed in a different notation. In this case, the transformation is applied to the complete program. For example, a compiler applies a transformation from a source to an object language program. Frequently, however, it is useful to perform transformations from a form of the program expressed in some notation to another form of that program expressed in the same notation. In this case, such transformations can be successively applied to selected parts of the program. When such transformations are applied to the program expressed in the target language (or source language for some object program), they are known as source-to-source program transformations (see, for example, Loveman [38]).

Certain program transformations are of frequent use in program development, and in some sense may be said to capture the programmer's knowledge about the kinds of ways he can proceed during the development (Gerhart [21]). If they can be expressed in a suitably concise form, they provide a useful unit for inclusion in a library of reusable program development concepts.

We wish to apply program transformations to the representation of data concepts during program development. In order that successive transformations may be applied to parts of the program that are concerned with each data concept in turn, the transformations will be between forms of the program expressed in the same notation. This notation must allow the expression of both the abstract form of data required in the initial abstract solution, and of the concrete form required in the target language.

2.3 Data Types for Abstraction

As a first step towards considering the representation of abstract data concepts in programs, we look at the role of the data type for expressing data abstractions.

Basic types are familiar from traditional programing languages. More recent languages allow programmer-defined types to be introduced in programs, and for their specification in terms of language-defined structuring methods.

2.3.1 Basic Types.

Data types provided as primitives in programing languages are familiar from languages such as Algol 60. There, the simple types integer, real and boolean are basic to the language, and allow the use of data items of those types without the need to consider the representation of the items in more primitive machine-oriented terms. Upon defining a variable as, for example,

<u>boolean</u> b;

then it is implied that variable b can only take the conceptual values true or false. It does not matter whether the variable is represented as a single bit or a complete word in the store of the machine on which the language is implemented or whether true is represented by a '0' or '1' value. Provided the implementation consistently supports the true-false abstraction, it can be ignored when programming in Algol 60. The language also provides certain basic operations that can be performed on booleans, such as 'and', 'or' and 'not', whose implementation can also be ignored, provided the results of applying the operations are consistent with the abstract notion of the operations' effects.

Similarly, reals and integers in Algol 60 have associated sets of values that they may take, and operations that may be applied to them. Each operation requires operands of a certain type, and a compiler can perform type checking in order to detect (in most situations) whether a programming error has led to an incorrect use of some data items.

For a given target language, we shall call data types that are directly implemented in the language <u>basic types</u>. So therefore, for example, real, integer and boolean are the basic types of Algol 60. Similarly we shall call the operations on such types that are directly implemented in a given language the <u>basic type operations</u> for that language. So, for example, division / is a basic type operation on reals in Algol 60.

2.3.2 Programmer-defined Types.

Languages providing a fixed set of data types, such as Algol 60, may allow a programmer to define his own 'operations' in the form of procedures. This has the advantage that since a procedure declaration in Algol 60 must generally specify the types of its parameters, including that of any result, type checking may be applied to the procedure calls.

However, the programmer cannot define his own data types in such languages, and thereby misses the opportunity to tailor the data concepts and the associated type checking to his own needs.

Two main approaches to the inclusion of programmer-defined types in programming languages may be distinguished. The first approach is characterised by the use of types in Pascal [28] and of modes in Algol 68 [57]. In these languages, the programmer may introduce an identifier to stand for a kind of data concept. This type identifier is used when declaring variables or specifying parameters in order to distinguish the type of data item involved; this enables type checking to be used to trap some kinds of programming error. When a programmer-defined type is declared in such a language, it must be specified to have a particular (often structured) set of values. The structuring methods allowed in the declaration are directly implemented by the language, so the type declaration therefore effectively fixes the representation used for the type to be that provided for the declared structure by the language. Operations allowed upon data items of the programmer-defined type are those provided by the language on the structure that the type is specified to have

Example: In Pascal, a programmer-defined type could be declared as

```
        type  hue = set of colour
```

This would fix the representation of data items of type 'hue' to be that provided for sets by the Pascal implementation (usually a bit representation within one machine word). It would also fix the operations that could be applied to data items of type 'hue' to be union, intersection, set difference and set membership.

The second approach to the inclusion of programmer-defined types in programming languages is that of the 'class' or 'abstract data type', illustrated in languages such as Simula 67 [10], CLU [35,33,34] and Alphard [60]. In these languages, the programmer not only introduces an identifier for his new type, but also defines the operations that may be applied to instances of the type. Programs may therefore be written using the abstract concept of the type without any knowledge of its representation. The representation of the type is defined by giving the structure of the data used to represent an instance of the type and by implementing the programmer-defined operations in terms of language-specific operations on the structured data. The representation, though partly depending on the language-implemented structuring methods as in the first approach, is essentially programmed individually for each of the programmer-defined types.

Example: In Simula 67, a programmer may introduce the data concept of a bounded queue of integers, with operations to join and leave the queue, and to test whether it is currently full or empty. Instances of the abstract type 'integerqueue' could be created:

```
ref  (integerqueue) q;
q :-  new integerqueue;
```

and used with the defined operations in writing the program:

```
- - - q.join (m); - - -
- - - if  q. empty then q. leavehead (n);- - -
```

The representation for the queue is defined in a class:

```
class integerqueue;
   begin integer front, back;
   integer array Q [1:20];
       procedure join (i); integer i;
          begin
          front:= mod (front, 20) + 1;
          Q [front] :=i;
          end;
   end integerqueue;
```

which uses the language-provided array structure, plus programmer-defined bodies for the operations, to give the desired implementation.

2.3.3 Type Specification.

Before a representation can be chosen for a programmer-defined type, the specification to which the representation must conform must be known.

Example: In the previous example, besides knowing the identifiers and parameters of the operations on the type 'integerqueue', it is also necessary to know the meaning of the operations before either a program can be written which uses them, or a representation can be chosen for the type.

A type specification may be left as an informal notion in the mind of the programmer, or an attempt may be made to document the specification with varying degrees of formality. In order to assist the selection of representations for the data concepts used in a program, some explicit form of specification is required.

Liskov and Zilles [36] describe various ways in which data abstractions may be specified. They identify five techniques:

i) use of a fixed formal domain of objects

ii) use of an appropriate, but otherwise arbitary, known formal domain

iii) use of a state machine model

iv) use of axiomatic definitions

v) use of algebraic definitions

The above categories are in increasing order of 'abstractness of specification', the earlier ones tending to exhibit more representational bias than the later ones.

For our purposes, a method of data specification was required which, while hopefully being largely representation-independent, was sufficiently capable of fitting into the program development process that useful aid could be provided in choosing data representations. The specification method used is most closely related to Liskov and Zilles second category, the use of an arbitary formal discipline, which, as they point out, is analogous to writing programs in a language which provides several data structuring facilities. We shall return to this approach in subsection 2.4.2.

2.4 Data Structures.

Although we wish to take a broader view of the term 'data structure', it is useful first to consider the kinds of structures that are used in conventional programming languages.

2.4.1 Structures in Programming Languages

In Algol 60, for example, the only kind of structure available for data is the array. This consists of a mapping from a particular subrange of the integers (or the Cartesian product of integer subranges in the case of multi-dimensional arrays) onto a basic type of the language (integer, real or Boolean). There is an operation associated with the array structure, namely that of indexing an element of the array.

Example: In an Algol 60 program an array could be declared as

<u>integer</u> <u>array</u> A[1:10]

in which case the indexing operation A[i], where $i \in [1,10]$, would select the ith element, of type integer, in the array.

The relative lack of data structuring facilities in Algol 60 led to the development of various Algol-like languages that tried to incorporate better facilities. For example, Algol W [58] includes records as a structuring method, allowing the Cartesian product of several different data types to be constructed and named and so define what is essentially the set of values of a new data type. As we described in 2.3.2, Pascal [28] makes such type definitions even more explicit, and introduces further structuring methods, including variant records and sets, whilst in the declaration of modes, Algol 68 [57] uses structures such as discriminated unions and flexible arrays.

The data structuring techniques available in these and similar languages all share a common property - the language provides a fixed method of implementation for each kind of structure. Once the programmer has written his program in terms of these structures his representational task is over, since the target language has been reached and no further representational choices need be made.

2.4.2 Structures for Specification.

Our approach to the representation of data uses data structures in a way that allows data concepts to be specified in terms of structures, but does not restrict their implementation as in a conventional programing language.

In order to specify an abstract data type, and its associated operations, we require that its (abstract) values are expressed in terms of a structure chosen from a certain set of abstract data structuring methods, and its operations are expressed in terms of manipulations applicable to the specified structure. The set of data structuring methods, and the fixed set of manipulations defined on those structures act as a formalism for the purposes of specification.

Consider, for example, the programmer-defined type 'complex', introduced as an abstract data type into a program dealing with complex numbers. The programmer might also define operations 'add' and 'subtract' on complex values, and so be able to write an abstract program that manipulates complex items, ignoring for the time being how those items are to be represented. Of course, when he uses the operation 'add' on two complex values in his abstract program, the programmer knows how the operation should behave, though he may not yet have thought about how he will actually implement it in terms of the target language. By giving a specification of the type 'complex', in terms of the logical structure by which it could be described, and the effects of the operations on such a structure, the programmer can define his conceptual view of the type that he has introduced.

Example: The type 'complex' and the operation 'add' might be specified as follows, using an ad hoc syntax:

<u>type</u> complex = (real_part: <u>real</u>; imag_part: <u>real</u>);

```
add (c1:complex, c2:complex): complex
 =  complex (c1.real_part + c2.imag_part,
                  c1.imag_part + c2.imag_part)
```

This denotes that a complex item is considered as composed of two components, both of type real, which can be selected by the identifiers 'real-part' and 'imag-part'. The 'add' operation is specified as taking two complex items and constructing a new complex item, whose first component is the sum of the real-part's of the operands, and whose second component is the sum of the imag-parts. In other words the type can be specified in terms of a Cartesian product structure, and the operations on the type can be specified in terms of operations associated with that structure (construction of an item from its components, and selection of an identified component of an item, in the above case).

Note that such a specification hasn't necessarily implied a particular implementation of the type, since the Cartesian product structure, or the operations associated with it, such as construction, need not be basic to the target language. We will discuss this further in a moment. First we define our usage of the term data structure.

A data structure is a means of specifying the way in which objects of a compound type are composed in terms of constituent objects of other types.

Each data structure is composed using one or more structuring methods. These are the type combinators from which complex structures may be built.

> Example: A Cartesian product is one kind of structuring method. In the above comple x number example, the logical structure of a complex number was defined using a Cartesian product combination of two real types.

A structuring method may be fotmally specified. Hoare [25] gives axiomatic definitions of all the structuring methods used in subsequent chapters. We shall concentrate on the use to which such structuring methods may be put rather than on their formal specification.

For each structuring method there are an associated set of operations which may be used to refer to or manipulate data objects defined to have the given structure.

> Example: In order to refer to the two components of complex data objects, selection operations were used above, and a new complex item was constructed from its constituent parts. Both the selection and construction operations are specifically associated with the Cartesian product structuring method.

In order to be able to help the programmer to choose representations for his data, we require that he specify the logical structure of the abstract data types in his program in terms of a given set of structuring methods (described in the next subsection). In addition, the operations that he requires on his abstract data types must be specified in terms of manipulations on those logical structures, using the given operations associated with structures (illustrated in section 4.3). Once the program has been expressed in terms of these structuring methods and their associated operations, representations may be chosen for the data structures so that they can be implemented in the target language.

2.4.3 Structuring Methods.

For the purposes of specifying the structure of data types a fixed set of structuring methods were used during the present investigations. They correspond closely with those given by Hoare in his 'Notes on Data Structuring' [25], and were chosen to allow many commonly encountered data structuring concepts to be expressed by a relatively small set of structuring methods. For example, the concepts of a stack, a list and a queue are all particular uses of the general structuring method of a sequence.

The structuring methods used are the following:

integer subrange, Cartesian product, discriminated union, array, set, sequence, recursive definition.

We have omitted enumeration from the techniques discussed by Hoare for the sake of simplicity. The transformation from an enumeration to an integer subrange is straightforward and of only syntactic significance.

Example (Card game): Rather than specifying the type of the suit of a playing card as

```
        type SUIT = (hearts, clubs, diamonds, spades);
```

we shall presume that each value of the type is mapped onto an integer and specify the type as

```
        type SUIT = 1..4;
```

This kind of representation will be presumed for all types that could have been specified as enumerations.

(strictly, enumerations and integer subranges do not specify structured types, and should not therefore be called structuring methods. However, in a situation where a structured type may possibly be represented in terms of an unstructured type, as in the example that follows in 3.4.2, the distinction is no longer so clear. We include integer subrange as a structuring method so that we can refer to the structuring method used in a type specification no matter whether the type is structured or unstructured in conventional terms).

The structuring methods that we use include most of those encountered in conventional programming languages. We presume for our present purposes that the target language towards which the program development is proceeding implements some restricted subset of the given structuring methods.

> Example: Pascal [28] implements all the given structuring methods, but only in a restricted form. A Pascal file, for example, is a restricted form of a sequence, and a set may only have a limited maximum cardinality, dependent on the word size of the implementation. Low-level assembly languages generally only 'implement' a single array structure, being the indexable storage of the machine itself.

When writing the intial problem solution in the form of an abstract program, the programmer may use any of the structuring methods in their most general form. The objective during the representational choice process will be to implement those structures which the programmer has in fact used in terms of the restricted structuring methods available in the target language.

The structuring methods are described informally in Appendix I, which also includes a description of the data operations associated with each method (to be discussed further in Chapter 4). Hoare [25] presents a more formal, axiomatic, definition of the methods. We shall illustrate the use of the structuring methods by giving the type specifications used in the abstract forms of the programs for the two example problems.

Birthdays example (type specifications):

```
type DAY = 1..366;
type PERSON = 1..5000;
type GROUP = set of PERSON;
type TABLE = array DAY of GROUP;
```

Types DAY and PERSON, both defined as integer sub-ranges, will be used for the day of the year of a birthday and the identification number of a person, as required in the problem specification. A type GRUUP is introduced, specified as a set of people, to denote a group of people born on the same day. The data to be stored in the program can then be defined as a type TABLE, specified to have one group of people for each day of the year. This is the abstract form of the data. In practice, the array will be sparse (since the problem specified a maximum of 100 people, so many days will have an empty group of people), and need not necessarily be represented as 366 contiguous components. The representation for the sets etc. has also still to be decided.

Card Game example (type specifications):

```
type PLAYER = 1..2;
type RANK = 1..13;
type SUIT = 1..4;
type PLAYING-CARD = (suit:SUIT; rank:RANK);
type HAND = set of PLAYING-CARD;
type PLAYERS = array PLAYER of HAND;
type FACE-UP-PILE = sequence PLAYING-CARD;
type FACE-DOWN-PILE = sequence PLAYING-CARD;
```

Two players are involved in the game, and a type PLAYER is specified as having the values 1 or 2 to denote each player respectively. The RANK and SUIT of a card are specified as integer subranges also. The type PLAYING-CARD can then be specified as a Cartesian product (or record) having components of types RANK and SUIT, and giving selectors in order that each component may be identified separately. A hand of cards is defined then as a set of playing cards. The data relevant to the states of both of the players hands of cards are denoted by the type PLAYERS, which defies one HAND for each player. The other cards in the game, not in either of the players hands, are in one of the two PILEs. Note that though both HAND and the PILEs contain PLAYING-CARD components, the first is specified as a set and the second two as sequences. This reflects the fact that the cards in a hand are not in any specific order, and will not be inserted and removed in a predetermined fashion, whereas in a pile the cards are ordered and will be added or removed from a specific place, such as the top of the pile. The two kinds of pile are specified as different types because their manipulation will be significantly different, and therefore a different representation may be required for each. We associate one representation per type, rather than one representation per instance of a type, so in order to allow different representations to be chosen, we must have a different type for each pile. This restriction will be discussed at more length in section 6.2.2.

2.4.4 The Representation of Structures.

Having defined the logical data structures required in the program, it remains to represent these in a suitable target language form. A structuring method such as a sequence or a set may not be implemented directly in the target language. In fact the kind of target language most likely to be used for the sake of efficiency may only include the array as a basic structuring method. It is therefore necessary to express data having, say, a sequence structure in terms of some representing data structure, and to express manipulations of the sequence data in terms of corresponding manipulations on the representing data.

Besides those data structures which use structuring methods which are not implemented in the target language, it is also necessary to represent complex data structures formed from a hierarchy of components each using a possibly different structuring method. In general, therefore, we wish to have representations not only for various data structuring methods, but also for compound data structures.

A data structure representation consists of the representation of a given (possibly compound) data structure in terms of some other (possibly compound) data structure and the implementation of each of the operations associated with the first structure in terms of the operation associated with the second structure.

> Example: A data structure consisting of a sequence of items of some type T may be represented in terms of an array of items of type T together with the index of the end of the sequence. The operation of inserting a new item of type T onto the end of the sequence is implemented in terms of updating the end index and inserting the item into the appropriate element of the array.

In the next chapter data structure representations are discussed in more detail, and in particular it is shown how they can be expressed as program transformations.

CHAPTER 3

EXPRESSING DATA REPRESENTATIONS AS TRANSFORMATIONS

3.0 Summary

In this chapter we go on to consider how program transformations may be used to express data representations.

First we look at some examples to illustrate the kinds of representations that we wish to express, and we discuss the factors involved in applying a transformational approach to describing such representations. The next sections in the chapter are concerned with transforming the data structure specification of a data type, and examples are given of how representations may be expressed and applied in these terms. For a full description of a representation, it is also necessary to consider the transformation of the operations upon the type being represented, fuller discussion of this being left to the following chapter. Finally in this chapter we consider the construction of libraries of data representations, and show how the transformational approach may assist in building libraries.

3.1 Examples of data structure representations.

Firstly in this chapter, then, we consider some examples of the kinds of data structure representations that may often be useful in programs. They illustrate the sorts of representations that could suitably be included in a library.

3.1.1 Set represented as a sequence.

Suppose that a set of similar items is required to be manipulated in a program.

> Example (Birthdays): In the birthdays problem, we require to store the set of people who have a birthday on the same day of the year, for each such day. As the input data is analysed, new elements will be added to the appropriate sets. To produce the required output, the elements of each set are listed.

A common representation for such a set is to use a linked list of items, one item for each current member of the set. When an item is to be added to the set, it will be linked into the list, possibly with a check to see if that item is already in the list. If an item is to be removed from the set, a search will be made for it in the list, and it will be removed by manipulating the relevant links. In fact, there are several ways of representing lists, with different linking arrangements, or even contiguously within a piece of storage.

A generalisation and abstraction of the list, namely the sequence, is one of the structuring methods adopted in 2.4.3 for specifying data types. The representation of a set in terms of a sequence is therefore a useful transformation to apply in the representational process. The choice of a subsequent representation for the sequence can be made independently.

3.1.2 Packed data.

Another example of data representation, usually only made directly available to the programmer of low-level languages, is that of packed data. This normally is required when it is necessary to store several data items, each of which will only take a small range of values, and where it would be inefficient to use one word of storage for each item.

Of course, describing packed data representation in these terms depends very much on factors such as the word size of the machine. However, it is possible to consider abstract counterparts to packed data representation in a machine-independent form, better suited to inclusion in a general library of representations.

> Example: Given two separate items i and j whose values can range over, say, the integers 1 to 10 and 1 to 20 respectively, it is possible to represent these two integer subranges in terms of a single subrange 1 to 200, with value $(i-1)\times 20 + j$ corresponding to given i and j values.

3.2 Applying a Transformational Approach.

When we wish to investigate applying a transformational approach to the representation of data, there are two main aspects to consider. Firstly, representations like those of the previous section must be expressed as transformations, so that they can be included in libraries etc. Secondly, it must be possible to determine whether a particular transformation is applicable to a given program, and, if so, how that transformation is to be implemented.

3.2.1 Expressing Representations.

In order to express the representation of one data structure in terms of another, as a transformation to be applied to a program, we require a suitable notation for the task. This notation will be used to write an initial version of the program containing abstract data concepts, will be used in the transformation descriptions to express the changes to be made to the program, will document the program at each stage of transformation, and will finally be used to express the target version of the program. It will include means for describing not only abstract and concrete data structures, but also the manipulations to be carried out on data items in programs which use such structures.

We wish to illustrate some of the factors involved in choosing and using such a notation. We have not considered in detail the design of a notation to meet these requirements, so that the subsequent use of specific notations is for purposes of illustration rather than a suggestion for a new language. Where possible we shall adopt a Pascal-like notation [28] in order that details irrelevant for our discussion need not be explained at length.

In expressing a data representation as a transformation, two parts may be distinguished. The <u>structure transformation</u> part describes how a data type specified in terms of a particular data structure may be represented by a different structure.

> Example: To describe the representation of 3.1.1, the structure transformation will express how a type specified to be a set may be represented instead as a type specified to be a sequence.

The <u>operations transformations</u> part describes how each of the operations applicable to the initial structure that are to be implemented by the representation may be transformed into operations upon the new structure.

> Example: For the representation of 3.1.1, an operation such as inserting an item into a set will be transformed into operations upon the sequence which represents that set.

3.2.2. Matching and Implementing Transformations.

Besides devising a suitable notation in which to express data representations as transformations, such transformations must be applied to a program in an appropriate fashion. Given an abstract data concept, specified as having a particular structure in a program, and given a library of possible transformations, there is firstly the selection of those transformations from the library that might be used in the representation of the data.

In order for a transformation to be applicable, the data structure, and the operations on the structure, which it transforms must match the specification and use of the data in the program.

> Example: For the set-as-sequence representation of 3.1.1, a transformation that expresses this representation will not match, and hence not be applicable to, a program if the representation does not include a transformation for an operation, such as removing an element from a set, which is used in the program on an item of the type being represented.

Given that a set of transformations may be matched as being applicable to a data type in a program, there remains firstly the choice of which of these, if any, to use in the program, and secondly the implementation of the chosen one. The selection of the transformation to implement involves an evaluation among the set that has been matched. The implementation of the chosen transformation is carried out by applying its constituent structure transformation and operation transformations to the program, giving a new version of the program to which further transformations may be applied.

3.3 The Structure Transformation

In this section we introduce a notation for describing the structure transformation part of a data representation. The structure transformation shows how a data type in a program, specified by a particular data structure, may be represented by a different kind of structure.

In section 3.4, the form that the structure transformation takes is described in more detail by considering specific examples. Here we introduce its general form.

3.3.1 The General Form.

The general form that a structure transformation takes in our experimental system is as follows:

```
        type specification          type specification
        for 'old' structure    =>   for 'new' structure
```

The 'old' structure is to be represented in terms of the 'new' structure, so that a data type which has the 'old' structure in a program may be transformed to have the 'new' structure.

Either or both of the structures may be simple or compound. A simple structure consists of a single type specification, and a compound structure consists of a related set of type specifications. Examples of each will be given in the next section.

In order that a given structure transformation be applicable to a given type in an abstract program, the specification of the type in the program must match that of the old structure. If the old structure is simple, only the specification of the type itself need match. If the old structure is compound, not only the specification of the type, but also that of its related subtypes in the program must match the compound structure. Matching is considered more fully in section 3.5.1.

3.4 Example Structure Transformations

We shall illustrate some possible forms of structure transformation by giving those that might be written for the two representations introduced in section 3.1, plus an addition more general kind of representation.

3.4.1 Set as sequence.

The structure transformation for this representation can be straightforwardly written as:

```
        type A = set of B;    =>    type A = sequence of B;
```

The type identifiers A and B stand for two distinct types in some program. The transformation expresses the fact that if some type A in a program is specified as being a set of components of type B, then if the set-as-sequence representation is implemented for type A, its specification is to be transformed to that of a sequence of components of type B.

In this case, both the old and new type specifications have a simple structure.

3.4.2 Packed data.

For this example, two integer subranges are to be represented by a single subrange. This is expressed as:

```
        type A = (s1:B; s2:C);
        type B = k..l:
        type C = m..n;          =>    type A = 0..(l-k+1)*(n-m+1)-1;
```

Here there is a compound old structure and a simple new structure.

The old structure specifies that the type to be represented, A, must be specified as a Cartesian product of two other types, B and C, each in turn specified to be an integer subrange. The bounds of the subranges for types B and C are shown as constant identifiers, so that they will match with whatever values the bounds for the corresponding types in a program may have. The new structure specifies that type A will be transformed into an integer subrange with lower bounds of zero and an upper bound given by an expression whose value may be determined once the appropriate values for k,l,m and n are resolved.

3.4.3 Indirect Representation.

A general representation that may be applied to any data item, but is particularly used for items of unknown or variable size, introduces a level of indirection into data manipulation. In this indirect representation, instead of storing a data item in a given place in store, which may lead to difficulties when the size of the item varies, a pointer of fixed size is stored in that position which in turn points to the actual item. The item itself is then stored in a suitably sized section of a storage area containing other items of the same kind. Usually some sort of storage allocation mechanism and possibly garbage collection must control the common storage area, allowing the size of the individual items to vary dynamically and yet the total storage required to remain under control. Indirect representations of this sort can be used, for example, for several sequences of varying length, or for recursive data structures where the depth of recursion may vary. They also allow techniques such as shared data items, where, if two items have the same value, they need only be implemented in terms of pointers to a common shared value (though, as will be discussed in section 4.1, this leads to problems with selective updating).

Note that we class such use of pointers as a particular choice of representation for the data concerned, not as an essential part of the definition of such data, as is required in many programming languages (for example, records and references in Algol W [58]) We wish to define sequences and recursive structures in abstract terms, so that the choice of representation is left open in order that a suitably efficient choice may be made from a set of alternatives.

An extension to the notation introduced so far is required for expressing representations such as the indirect representation. This is because these representations require some 'global storage'. The notation used so far has only allowed the description of representations where there is a one-for-one change in the structure for each item of the type being represented. Each item of the type having the old structure is implemented in terms of the new structure. However, in some cases, such as indirect representation, besides this change in the form of each individual item, there is also a need for some data that is common to all items of the type being represented and is therefore global to the range in which items of the type may occur.

The following structure transformation demonstrates the notation used in the experimental system to cater for this and other extensions to the notation presented so far.

```
type A = anystruc;
    =>  type A = 1..MAXNUM (A);
        type U = (lastused:A; s:STORE);
        type STORE = array A of B;
        type B = anystruc;
        global uu:U;
```

The specification of the old structure in this transformation consists of the keyword anystruc, which stands for any structuring method. The representation can therefore be applied to any type, no matter what its structure specification. Wherever the keyword is repeated in the new structure, in this case in the specification of type B, it will correspond to that of the old specification of type A. When the transformation is applied to a program, whatever structure type A has before the transformation will be used by type B after it.

The new structure given to type A in the representation is an integer subrange, where the upper limit to the subrange, MAXNUM (A), is a special value. It denotes the maximum number of data items of type A that will be used in the program, and is one of a set of type properties that are maintained in the experimental system and may be used in writing representations. A full list of the properties are given in Appendix I.

At the end of the specifications of the types on the right hand side of the transformation there is a declaration of the global part of the representation. This consists of a variable 'uu' of type U, declared using global rather than var, which is to be added as a global variable in the program when the representation is implemented for a type. (The preservation of the uniqueness of identifiers when transformations are applied will ensure that a different global variable is used for each type that might use the indirect representation). The type of the global component U is specified to consist of an indication of the 'nextfree' item of the new type A, and a storage component made up of an array of elements each having the structure of the original type A. In other words, storage is made globally available to store the maximum possible number of items of the type being represented, and this storage is managed by keeping track of the next free array element that may be used by the program. Items of the old type A are now represented as an index to the element in the storage at which its value is held, effectively acting as a pointer to the stored value. The representation therefore allows sharing of pointers in the conventional manner of indirect referencing.

This is only one of the possible ways of handling an indirect representation. Other examples could be written which deal with storage allocation in a more sophisticated fashion.

3.5 <u>Applying structure transformations to programs</u>

In order to apply a data representation to a type in an abstract program, the representation must first be found to match with the given type, and then the program must be transformed to reflect the new representation. We shall consider the matching and transformation for the structure transformation in this section, leaving the additional consideration of the operation transformations to the following chapter.

3.5.1 Matching structures.

In order for a structure transformation to be applicable to a given type in an abstract program, the old structure specification in the transformation must match with the specification of the type (and any necessary subtypes of that type, if the old structure is compound) in the program.

Example: Given a structure transformation such as that for the set-as-sequence representation:

<u>type</u> A = <u>set</u> <u>of</u> B; => <u>type</u> A = <u>sequence</u> <u>of</u> B;

the identifiers A and B in the old structure specification may be matched to specific type identifiers in a program only if those identifiers are specified such that the first type is a set of components of the second type. If in the program we have the specifications:

```
type HAND = set of PLAYING-CARD;
type PLAYING-CARD = (s:SUIT; r:RANK);
```

then the above structure transformation may be matched for the type HAND, with A and B being matched to HAND and PLAYING-CARD respectively.

For a compound structure specification, the types can be regarded as forming a directed graph with type identifiers at the nodes and with a link from one type to each of the other types used in its specification. So, given an old structure with the compound specification:

```
type A = (s1:B; s2:C);
type B = set of D;
type C = array E of D;
```

the following directed graph is obtained:

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & & \\
B \arrow[dr] & & C \arrow[dl] \arrow[dr] & \\
 & D & & E
\end{tikzcd}$$

In order that this specification matches that of a type to be represented in a program, the corresponding portion of the program's type graph must be isomorphic in structure to the above, as well as having corresponding structuring methods. So given the following type specifications in a program:

```
type T1 = (x:T2; y:T3);
type T2 = set of T4;
type T3 = array T5 of T6;
```

this gives the directed graph:

$$\begin{tikzcd}
 & & T1 \arrow[dl] \arrow[dr] & & \\
 & T2 \arrow[dl] & & T3 \arrow[dl] \arrow[dr] & \\
T4 & & T5 & & T6
\end{tikzcd}$$

which is not isomorphic to the previous graph. Hence type T1 cannot be matched with the old structure specification of the representation. If in the program type T3 was specified differently as:

```
type T3 = array T5 of T4;
```

then a match would be possible, since the graphs are then isomorphic as well as the structures of the specifications corresponding.

As well as identifying the correspondences between the program types and the old structure specification types, the matching may also be used to fix other identifiers in the old structure, such as subrange bounds.

Example: Given the old structure specification for the packed data representation of subsection 3.4.2:

```
type A = (s1:B; s2:C);
type B = k..l;
type C = m..n;
```

and given the following type specifications in
a program:

```
type PLAYING-CARD = (s:SUIT; r:RANK);
type SUIT = 1..4;
type RANK = 1..13;
```

then type PLAYING-CARD may be matched with the old
structure specification with correspondences set
up as follows:

Old structure	A	B	C	s1	s2	k	l	m	n
Program	PLAYING-CARD	SUIT	RANK	s	r	1	4	1	13

3.5.2 Transforming structures.

Once a data type in a program has been matched with a particular
data representation, and it has been decided to implement that
representation for the type, the program is transformed to meet the new
representation. In the case of the structure part of the representation,
the structure specification of the type is transformed from the old
structure to the new structure. All identifiers in the new structure
which have had correspondences established with program values in the
matching stage will be replaced by their program value. Any additional
identifiers in the new structure are incorporated into the program
taking care that they do not clash with existing program identifiers.
In the experimental system, each such identifier is modified by appending
a pair of digits unique to that implementation of the representation,
thus ensuring freedom from identifier clashes (assuming identifiers with
this form were not used in the original abstract program).

Example: Continuing the example given in the previous subsection for the packed data representation, the new structure specification for this representation is:

```
    type A = 0.. (l-k+1)*(n-m+1)-1;
```

On replacing identifiers with their program correspondences, and evaluating the arithmetic expression, the transformed specifications for the types in the program are:

```
    type  PLAYING-CARD = 0..51;
    type  SUIT = 1..4;
    type  RANK = 1..13;
```

3.6 Constructing a library of Representations

When contemplating the construction of a library of data representations from which choices can be made for implementation in a program, several factors must be considered. The feasibility of constructing a useful library depends largely on the number of representations that need to be included, and the relative completeness of the ones that are included. Clearly it is impossible to enumerate all possible representations, just as it is impossible to enumerate all possible programs. The question remains as to whether a 'useful' set of representations may be expressed in a manageably sized library.

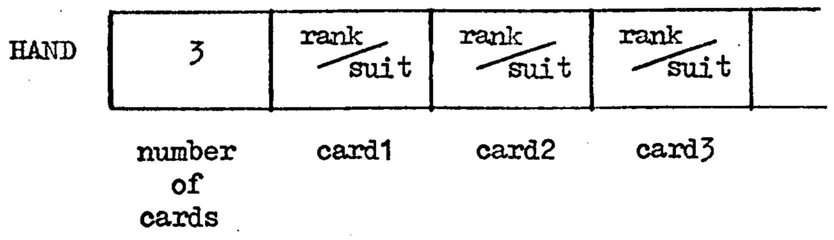
We shall consider how the transformational approach may help reduce the size of the library required, and discuss the completeness of representation libraries. As an example of the kind of transformations that could be incorporated in a library, we summarise those used in the experimental system.

3.6.1 Sequences of transformations.

In our approach to the representation of data, it is necessary to apply successive transformations to the program, until eventually the data is specified only in target language terms. This approach contrasts to that of, say, Low [39], who considers a data representation to take a data construct in an abstract program directly to a target language implementation.

The advantage of considering the separate individual transformations as only a partial step from the abstract to the concrete, is that different transformations may be combined into sequences in many varied ways. It is therefore possible to achieve a greater range of representations of the abstract in terms of the concrete program with relatively few simple individual transformations.

Example: Consider the diagrammatic representation for a hand of playing cards that was suggested in 1.2.2 as below:



where each playing card is packed into one array element, and the first element of the array denotes the number of cards in the hand. This representation can be achieved by a series of transformations. Given the initial type specifications for the hand:

```
    type HAND = set of PLAYING-CARD;
    type PLAYING-CARD = (r:RANK; s: SUIT);
    type RANK = 1..13;
    type SUIT = 1..4;
```

We can first choose to apply the set-as-sequence representation of 3.4.1 to obtain.

```
    type HAND = sequence of PLAYING-CARD;
    type PLAYING-CARD = (r:RANK; s:SUIT);
    type RANK = 1..13;
    type SUIT = 1..4;
```

(where we only show the transformed structure, though corresponding changes would be made to the rest of the program as well). Next we apply a transformation which represents the sequence as an array of contiguous elements with an indication of the current length of the sequence (and we assume a maximum sequence length of 52):

```
    type HAND = (length:A; elements:B);
    type B = array C of PLAYING-CARD;
    type A = 0..52;
    type C = 1..52;
    type PLAYING-CARD = (r:RANK; s:SUIT);
    type RANK = 1..13;
    type SUIT = 1..4;
```

Then we can choose the packed representation of

3.4.2 for the cards:

```
type HAND = (length:A; elements:B);
type B = array C of PLAYING-CARD;
type A = 0..52;
type C = 1..52;
type PLAYING_CARD = 0..51;
```

Finally some further transformations can be applied which represent the subrange values for PLAYING-CARD and A as the basic type int, and amalgamate the length field into the array to give:

```
type HAND = array D of int;
type D = 1..53;
```

The separation, as demonstrated in the above example, between say the transformation of a set into a sequence, then the transformation of the sequence to something else, allows the set to use all the possible sequence representations through the inclusion of only one transformation in a library. In a similar way, only a single version of the packed data representation may be used in combination with other transformations in many varied transformational sequences.

3.6.2 Completeness of a library.

It is clear that no library of data representations can ever be complete in the sense that it encompasses all possible representations, since new representations are continually being devised. A library, such as the one used in the experimental system for these investigations, is inherently extensible so that newly-developed representations can be added as required.

The number and kind of representations that should be included in a practically useful library would largely have to be determined through experience. To a certain extent, the representations that might be found worth including could be dependent on the nature of the programs in which they were to be used and the target language in which the concrete programs were to be written.

> Example: If the target environment allowed bit-addressable manipulations, it would probably be necessary to include the representation of sub-ranges in terms of varied length bit sequences as well as the fixed <u>int</u> type.

3.6.3 An example library.

The library of representations that was used during the experimentation was devised to include examples of many common representations. It probably does not, however, include a sufficiently varied range of representations for a practical library. For example, only one hashing algorithm is included, when a choice of several might be desirable.

A full listing of the representations in the example library is given in Appendix III. Here we shall summarise the various representations to give an idea of the range available.

Those concerned with subranges, products and unions:

SUBREP	subrange represented as a basic <u>int</u> type
SUBCP	Cartesian product packed into a single subrange
DU1	discriminated union with null alternative packed into a single subrange
DU3	union with null alternative given a separate tag field
SUBDU1	disjoint union packed into a single subrange
SUBDU2	non-disjoint discriminated union packed into a subrange

Those concerned with arrays:

ARRAY	sparse array expanded into a non-sparse array
BITS	array of bits represented as array of <u>int</u>
HASH	sparse array hashed into non-sparse array
SPARRAY	sparse array as default plus sequence of non-defaults

Those concerned with sets:

POW1	set represented as array of bits
POW2	set represented as sequence

Those concerned with sequences:

SEQ1	sequence as contiguous array with first element pointer
SEQ2	sequence as singly linked list
SEQ3	sequence as doubly linked list
SEQ5	several sequences as singly linked lists in common storage
SEQ6	sequence as contiguous array with last item marker

Others:

INDIRECT	all instances of a type in common storage with pointers

In addition, there are various transformations that act as equivalences between different structures (such as a cartesian product with two components of the same type, and an array with two elements of that type), whose application will be discussed in section 5.4.

CHAPTER 4

DATA OPERATIONS EXPRESSED AS TRANSFORMATIONS

4.0 Summary

In this chapter we turn our attention from the structure part to the operations part of data representations expressed in terms of program transformations. In the operations part of a representation, data operations applicable to the 'old' structure are transformed into operations on the 'new' structure.

When expressing the manipulation of data, and in particular when writing operations on data structures, certain problems arise that must be considered before a satisfactory means of writing representations for the data can be devised. We look at these problems in the following section, and then go on to draw an important distinction between data references and data values to clarify the concepts involved. This leads to the introduction of a classification of data operations, and the associated notation which was used in the experimental system.

The general form and some examples of operation transformations are then given, and finally the application of such transformations to programs is discussed.

4.1 Problems in Expressing Data Operations

Firstly, then, we consider some of the factors involved in data manipulation, and look at the sorts of problems that arise particularly when operating upon abstract data whose representation has yet to be chosen. We wish to be able to write an abstract program that uses the abstract data types declared by the programmer and expresses the problem solution in a data representation-independent fashion. The operations on the data used in the abstract program must however be transformable into appropriate operations on the final concrete data as the representations to be used for the data are fixed.

4.1.1. Representation dependence.

In a conventional programming language, each kind of data structure is represented in a particular fixed way, so that the operations available on the structures (such as indexing an element of an array, or selecting a component of a record) are implemented to suit the representation. However, when manipulating data structures whose representation has not yet been decided, the programmer does not yet know which operations will be efficiently implemented. Indeed the choice of the representation will be made in order to provide suitably efficient implementation of the operations required in the abstract program.

The operations that may be used on data structures when writing abstract programs must therefore be a sufficiently general set that the programmer can express the abstract manipulations to be carried out, but must also allow for the selection of representations for the structures which perhaps only implement a restricted set of general operations.

The selection of an appropriate general set of data structure operations is a complex task. The set of operations we have used in the experiments is not necessarily intended to be a complete or sufficient set, but does illustrate the need for operations not normally distinguished in programming languages.

4.1.2 Need to express unconventional operations.

As an illustration of the kind of data structure operation that appears necessary in a general set of operations, consider an array structure. If we declare a type and a variable of that type as follows:

```
type A = array 1..10 of 1..8;
var a:A;
```

then the conventional operation on a variable of an array structured type is that of indexing

$$a[i]$$

All changes in value of the array variable are made by selective updating of its components, a technique which suits the conventional representation of allocating one array element per storage cell.

However, if we wish to represent an array (which, say, is sparse, having few element values differing from a common default value) as a default element value plus a list of those elements whose values differ from the default, then other array operations become desirable. For example, this representation can easily implement an operation which sets all array elements to have the same value. Such an operation could also be implemented for the conventional representation as a loop which assigned the common value to each element in turn.

A note of caution is necessary, however, to check that operations which are too special-purpose, even though efficiently implemented by a particular representation, are not allowed to produce an over-large set of data operations. It is not clear what criteria should be used in judging whether an operation is of sufficient general use to be made available in writing abstract programs.

4.1.3 Need to distinguish references and values.

Not only are additional operations required beyond those conventionally used on data structures, but the existing operations are often too loosely used in programming notations for direct adoption when expressing data representations. For example, returning to the array of the previous subsection, the indexing operation has a different meaning, and hence may require a different implementation, depending on its context. When used on the left hand side of an assignment

$$a[i] :=$$

it denotes a reference to an array element, whereas on the right hand side

$$:= a[i]$$

it denotes the value of an array element.

In the case of the conventional array representation, with one element per storage cell, an indexed element has both a reference, enabling it to be updated, and a value. However, say we used a representation which packed the complete array into one word of storage, as we could indeed do for the above array, taking three bits per element for the ten elements. In this case there is no way of individually obtaining a reference to an element of the array (presuming a conventional target language or machine which does not allow bit-addressing).

For this reason we shall draw a clear distinction between data references and values in order to distinguish data operations more clearly.

4.2 References and Values

In order to clarify the different operations that may be applied to data items, and to identify those operations that a given representation can implement, we shall make the distinction between data references and data values more explicitly than in most programming languages. This distinction, at its most basic, corresponds to the distinction between the address of a cell in a computer store and the contents of that cell. We shall consider it in those terms first, and then generalise to its use when dealing with abstract data that has not yet been represented in terms of storage cells.

4.2.1 Storage derivations.

The distinction between references and values, particularly in the context of programming languages, arises when relating the semantics of programs to the underlying implementations in storage (see, for example, Barron [5] , Lomet [37] and Walk [52]). Take, for example, the case of two variables in Algol 60, declared as:

```
integer i,j;
```

When the block containing this declaration is entered during the execution of the program in which it occurs, two storage cells will be allocated, one for each variable. These will be used to hold the values of the variables as they change during the subsequent manipulations. These storage cells are referred to by their address in storage, and when they are allocated, a conceptual relation is set up between the variable identifier in the program and the address of the associated cell in store. When, therefore, the variable is manipulated in the program, the relevant cell in the language implementation can be located and manipulated accordingly. Possibly the most common form of data manipulation in Algol is assignment, so consider the assignment statement within the block containing the above declaration:

```
i := j;
```

In storage terms, this means that the <u>contents</u> of the cell corresponding to the variable j should be stored at the address of the cell corresponding to the variable i. In other words, the variables must be interpreted to mean different things, the location of a cell or the contents of a cell, depending on whether they occur on the left-hand or right-hand side of the assignment statement. Strachey [50] pointed out this distinction and used the terms L-value and R-value to refer to the different results of evaluating the expression on each side of an assignment statement. In Algol 60, besides a simple variable identifier, many different kinds of expression may be written on the right-hand side of an assignment, for example:

```
i := j+1;
i := if j=0 then 0 else j+1;
i := a[j];
```

(assuming 'a' is declared as an integer array). Each right-hand side expression evaluates to an R-value, in other words an integer value in these cases. However, on the left-hand side, Algol 60 only allows one kind of expression besides the simple variable identifier, namely the array subscription:

```
a[i] : = j;
```

Here, the left-hand side must be evaluated to produce an L-value, in other words the address of the array element that is to have a new value assigned to it. One of the early languages that was developed based on Algol 60, CPL (Barron et. al. [4]) extended this idea to allow more general L-value expressions. For example, the following assignment could be written in CPL:

```
(i>j→i,j):= 0;
```

meaning that either i or j was to be assigned the value 0, depending on the value of the relational expression i>j. In other words, the conditional expression.

```
i>j→i,j
```

returns the address of an integer variable as a result in this context, but would return an integer value as a result if placed on the right-hand side of an assignment.

To try and overcome the ambiguity of interpretation, some languages have adopted a specific syntax to show the action of taking the contents rather than the address denoted by a variable identifier. For example Bliss [59] uses a dot notation '.' to denote the 'contents' operation, so the above examples would be written:

```
i := .j
i := .a [.j]
a [.i] := .j  etc.
```

Algol 68 [57] uses the terms 'name' and 'value' instead of left-hand value and right-hand value, and allows the name of a data item to be a manipulable value itself, of mode <u>ref</u> M, M being the mode (or type) of the item. This allows a large scope for manipulation of references, and the dangerous potential of many levels of indirection, which with the complex coercion rules of the language can lead to difficulty in understanding the action of programs (see for example Hoare [27]).

4.2.2 Definitions.

We shall adopt the terms 'reference' and 'value', but will not consider references as manipulable items in the Algol 68 sense. We prefer to use these terms rather than, say, 'address' and 'contents' since we wish to consider abstract data structures and their representation. Real storage concepts such as addresses, which are essentially low level concepts, will not be introduced until necessary at an appropriate stage in the representation process.

So now consider the reference-value distinction in a general representation-independent sense. A data reference of a given type denotes a piece of abstract storage that may contain a data value of the associated type. (The reference may eventually map onto a specific addressable storage location, or may not, depending on the representation chosen for the type in question). Nothing is assumed of the abstract storage (i.e. it is not necessarily of fixed or known size, or contiguous or composed of uniform component cells etc.), except that it may hold any value of the associated type. As representations are chosen for the type and any of its components, the actual form of the storage will be decided, so that the final program will use explicit target language storage concepts.

4.2.3 Assignment.

Given a reference to a particular data item, a specific value may be set in the abstract storage denoted by the reference using an assignment operation. At any moment the current value associated with a given reference may be derived by a 'contents' operation. A reference to a data item is established either when storage is initially allocated for the item (e.g. on entering the block in which a variable is declared) or when a new data item is inserted into a data structure (e.g. inserting a new item into a sequence). Let us consider this for particular examples, firstly for a variable of an unstructured type, then for a structured variable.

If the following unstructured type has been declared in an abstract program

```
                    type RANGE = 1..20;
```

a variable of the type may be declared as

```
                    var r: RANGE;
```

Conceptually, this implies that if the abstract program were executed, upon encountering the declaration during execution some abstract storage must be allocated sufficient to hold any of the values of type RANGE (i.e. 20 distinct values) but not necessarily one word of concrete storage, or even 5 bits of concrete storage, will be allocated, since this will depend on the representation of type RANGE chosen subsequently. When the abstract storage is allocated, a reference is established to the storage, and an association is set up between the variable identifier 'r' and the reference. When 'r' is then used in the program, it will be taken to stand for the variable. So, upon execution of a statement such as

```
                    r : = 3;
```

the assignment takes the reference to the storage denoted by 'r' on the left-hand-side, and the value of the expression on the right-hand-side, and sets the contents of the storage to be the given value. In order to access the current value of the variable, given the reference to the storage in which it is stored, a 'contents' operation can be used, which we shall denote by the postfix operator '@' . For example, the statement:

```
                    r : = r@ + 1;
```

accesses the current contents of the variable whose reference is denoted by 'r', adds one to that value, and assigns the result back to the variable's storage. In a program expression such as this, it is possible to infer the existence of the contents operator, and therefore omit it in the program text to conform to normal Algol-like syntax:

```
r := r + 1;
```

However, the operation must still be carried out, and we may require that it have a particular implementation specified for it in the operation implementation section of a representation.

4.2.4 Structured variables.

Now consider the declaration of a structured variable in an abstract program:

```
type PAIR = (first : RANGE; second : RANGE);
var  p : PAIR;
```

On encountering the variable declaration during program execution, abstract storage must be allocated to hold any value of type PAIR (i.e. 20 x 20 = 400 different values). Again, whether these values are eventually stored with one word for each component, or packed into a single word, will depend on the representations chosen for types PAIR and RANGE. A reference to this abstract storage is fixed upon allocation, and denoted by the variable identifier 'p' in the program. But, since we wish to be able to selectively update the components of p, references also are established to the two components of p, and may be used on the left-hand-side of assignment statements. These component references may be derived from the reference to the complete variable, denoted by p, by a reference-selection operation:

```
.first
```

So, to selectively update the 'first' component of the variable, we may write:

```
                p.first : = 4;
```

(the notation is presented more fully in the next section). Given the reference to a component, its current value may be derived by the 'contents' operator, as for the unstructured example, so we may write statements such as:

```
                p.second : = p.first@;
```

(again, the '@' may be omitted and inferred from context, since the expression is on the right-hand-side of an assignment, and therefore must denote a value rather than a reference). But as well as taking the contents of an unstructured component of a structured variable, we also wish to be able to obtain the (structured) value of the whole variable from its reference by the contents operation:

```
                p@
```

In fact, if the type PAIR is represented in terms of a single word of concrete storage, by packing both components into a single cell, it may be impossible to individually reference the components within the word in order to selectively update them. In which case, it is necessary to obtain the value stored in the cell, in other words the structured PAIR value, and perform 'value-selection' operations to obtain the component values. Since in the abstract program we do not yet know how the type will be represented, we cater for both reference-selection and value selection, transforming between the two as necessary when a particular representation is to be implemented. So, given a structure value of type PAIR, the value of its 'first' component may be obtained by a value-selection which uses the notation:

```
                'first
```

which can be applied to the complete value of the variable p as follows:

```
                p@'first
```

In these examples, we have already begun to introduce the notation for data operations used in the experimental system. The next section classifies such operations and introduces the syntax developed to distinguish between the various kinds of operation.

4.3 Classification of Data Operations

Having shown the need to distinguish between data references and data values when considering data representations, and also shown the need for unconventional data operations, we now try to clarify the possible data operations. In this section we classify the operations according to their operands and their semantics. We also introduce a notation for textually distinguishing the different kinds of operation when writing programs or writing operation implementations in representations. The notation is not put forward as a proposal for inclusion in an implemented programming language, but as an aid to understanding certain data manipulations. It will be used in the subsequent examples of data transformations.

4.3.1 Ownership of operations.

A fundamental notion that we adopt is that each data operation has one and only one associated data type, which 'owns' the given operation. This is because, when a representation is chosen for a data type having a particular structure, it is necessary to implement each of the operations on data items of that type in terms of operations on the representing data. Thus it is necessary to know which operations must be elaborated when a particular data type is represented, namely those operations owned by the type in the abstract program.

The ownership of operations by a particular type is similar to the Simula class concept and its derivatives mentioned previously. However, there is an important difference between the use in classes and the use required here. In Simula, the operations in a class are associated with an instance of the class, rather than the class as a whole. In other words, conceptually each data object has its own set of operations different from those used on other objects, even though the objects may belong to the same class. Syntactically in Simula, each operation is prefixed by the object instance to which it must be applied; so each operation has a unique parameter of an object of the owner class. This makes operations, such as equality, between two objects of the class inherently asymmetrical. In Simula this would be written:

a . equals (b)

Here the operation 'equals' belongs to the object instance 'a', whereas the other object 'b', of the same class as 'a', is relegated to be a subordinate parameter. By saying that the operation 'equals' is owned by the class (or type in our notation) of the two compared items, each item can be a parameter of equal importance:

a = b

4.3.2 Operation categories.

We classify all data operations into eight categories, and give a particular distinguishing syntactic symbol to most of them in the notation. Each category contains operations with a common parameter specification and semantic interpretation.

	Category	Symbol	General syntax	Result
1.	Update	:	xr:op(params)	
2.	Reference-select	.	xr.op(params)	yr
3.	Contents	@	xr@	xv
4.	Value-select	'	xv'op(params)	yv
5.	Contruction	#	X#op(params)	xv
6.	Attribute	?	xv?op(params)	zv
7.	Relation		e.g. xv1=xv2	lv
8.	Iteration		e.g. <u>for</u> yv <u>in</u> xv <u>do</u> S	

Where: X is the owner data type of the operation.
xr is a reference to an item of type X.
xv is a value of type X.
op is the operation name.
Y is the type of a component of X (hence yr,yv).
zv is a value of a standard type such as <u>logical</u> or <u>int</u>.
lv is a value of type <u>logical</u>.
params are a set of value parameters.

<u>Figure 4.1</u> <u>Operation Categories</u>

The distinguishing characteristics of each category are listed in Figure 4.1. We shall here briefly summarise the intent of each category, and in the next subsection will give examples of data operations which use the suggested notation.

Each data operation is applicable to data references or values of its owner type, and may take values of other types as parameters and return references or values of various types as a result. Each category of operation takes a particular combination of 0,1 or 2 references or values of the owner type, as shown in the general syntax column. Some kinds of operation may take a bracketed parameter list of one or more additional data values of types which depend on the particular operation. The characteristic symbol distinguishes the category by the syntax for six of the operation kinds, the relation and iteration having their own individual syntax. Within a category each operation has its own individual name.

An update operation acts to change the value associated with the data item whose reference is given. It is the only operation which causes a change in the stored data values. Assignment is the most common form of update operation, whose operation 'name' is the symbol =, and which requires one additional value parameter of type X which is the value to be assigned to the referenced data item.

The two kinds of selection, reference-select and value-select have been briefly introduced in section 4.2, as has the contents operation.

A construction operation constructs a new value of the owner type, generally taking a list of parameters to do so. It may also be used to construct, say, an empty set or sequence, in which case no parameters are required.

An attribute operation is used to obtain, for example, the length of a sequence, or whether a set is empty or not. It differs from a value-select in that the latter selects a distinguishable component of a structure, of a type depending on the structure specification. The result of an attribute operation is of a standard type such as integer or logical. Whereas a value-select usually has a counterpart reference-select operation, the attribute operation has no such reference-returning counterpart.

A *relation* is the conventional comparison operation between two values of the owner type.

An *iteration* operation is not conventionally considered as an operation related to a particular data structure. However, some iterations, such as repeating some action for each item in a sequence or set, clearly depend for their implementation on the representation of the structure concerned. It is therefore necessary to consider how such an iteration is to implemented when representing the data structure, and so the operation belongs in a category of the data operations.

4.3.3 Example operations.

In order to illustrate the semantics and syntax of the various operation categories, we give several examples in this subsection. A more complete description of the operations associated with each of the structuring methods that were used in the experiments is given in Appendix I.

For the following examples, we presume the existence of various types and variables given by the following declarations:

```
type A = set of B;
type C = array D of E;
type D = 1..100;
type E = (s1:F; s2:G);
var  a: A;
var  b: B;
var  c: C;
```

By declaring variables in this way, a variable identifier such as 'd' may be used in a program to denote a reference to the data item of type D that is allocated by the declaration. The value of the item is obtained by the contents operation

```
d @
```

which may be used, say, in an arithmetic expression. A new value may be assigned to the variable in the conventional way

```
d := 8
```

using an update operation (where the brackets around the single parameter '8' are omitted to give the conventional assignment syntax). Tests on the current value of the variable use the conventional relation operations such as

```
d@ = 6
```

In the case of structured variables, such as the array 'c', elements of the array are obtained by selection operations. Reference-selection such as

```
c.index(7)
```

takes a reference to the variable 'c' and returns a reference in this case to the element indexed by the value 7. A similar value-selection

```
c@'index(7)
```

takes a value of type C i.e. 'c@' and returns the value of the seventh element of the array. Since the elements of the array are themselves structured as a Cartesian product with two components selected by the selectors s1 and s2, it is possible to concatenate selection operations to obtain, say a reference to the second component of third array element

```
c.index(3).s2
```

or the value of the first component of the last array element

```
c@'index(100)'s1
```

Turning now to the set structured variable 'a', as well as possibly using assignment operations on such a variable, other update operations allows elements to be inserted into or removed from the set, for example

a:insert(b@)

inserts the value of variable 'b' into set 'a'. Attribute operations may be used to test whether a particular value is currently an element of the set value

a@?has(b@)

or whether the set is empty

a@?empty

or determine the number of elements in the set

a@?numel

In addition, it is possible to construct a new empty set value

A#empty

so that, using an assignment, the set variable can be made empty

a := A#empty

Finally, it is possible to repeat some action 's' for each element of the set in turn

<u>for</u> bv <u>in</u> a@ <u>do</u> S

where the identifier bv may be used in S, and for each iteration will dénote the value of a different element from the set.

4.3.4 Example program.

As an illustration of the use of the notations introduced so far, we give the abstract program written for one of the two examples introduced in the first chapter.

Birthdays Example: Figure 4.2 gives the program as input to the experimental system. The type definitions are as explained in section 3.3.1. Three variables are declared; the first is the table of people having birthdays on given days, the others being auxiliary variables required in the execution. In the executable part of the program, the table is initialised so that each of its components is an empty set. In the input phase, controlled by the while loop, the identity number of each person in turn is inserted in the set indexed by their birthday. In the output phase, two nested loops, both being iteration data operations, cycle first through each day, and secondly through each element in the chosen set, in order to write the identity numbers of those people with birthdays on each day.

The abstract program for the other example program is listed in Appendix II, and provides a further illustration of the notation.

```
TYPE TABLE=ARRAY DAY OF GROUP;
TYPE DAY=1..366;
TYPE GROUP=SET OF PERSON;
TYPE PERSON=1..5000;
VAR T:TABLE; VAR D:DAY; VAR P:PERSON;
  BEGIN
  T:=TABLE#ALL(GROUP#EMPTY);
  READ(P);
  WHILE P¬=0 DO
    BEGIN READ(D);
    T.INDEX(D):INSERT(P);
    READ(P);
    END;
  FOR D DO
    BEGIN WRITE(D);
    FOR PP IN T'INDEX(D) DO WRITE(PP);
    END;
  END
```

Figure 4.2 Abstract program for Birthdays example.

4.4 Operation Transformations

We have already described the transformation of the structure part of a representation in section 3.3. In this section we describe the second part of the representation, the operation transformations. The structure transformation shows how the type being represented has its structure specification modified from the old to the new structure. The operation transformations show how operations on the old structure are modified into operations on the new structure so that the desired manipulation of the data is still carried out.

> Example: The structure transformation for the packed data representation, given in 3.4.2, showed how a Cartesian product of two integer subranges could be represented as a single subrange. It is necessary to also express operation transformations so that, for example, changing the value of a selected component of the old structure is transformed into an appropriate change in the value of the new structure.

We shall describe the general form of the operation transformation and then go on to illustrate some specific transformations.

4.4.1 The general form.

The general form taken by an operation transformation is similar to that for the structure transformation:

operation on 'old' structure	=>	operations or piece of program acting on 'new' structure

Just as the structures may be simple or compound, so the operations upon them may be simple or compound. A compound operation consists of a continguous set of data operations, and will be illustrated in the following subsection. The implementation of the operation, given on the right hand side of the transformation, may consist of a single or compound operation, or may even be a piece of program carrying out a series of operations on the new structure.

For each representation there may be a number of operation transformations, one for each of the operations on the old structure that the writer of the representation wishes to support on the new structure. In order that a representation be applicable to a data type in a program, each of the operations on that type that are used in the program must be implemented by the representation. In other words it is necessary that each operation owned by that type in the program is matched by an operation on the left hand side of one of the operation transformations for the representation. Matching and transforming operations is considered in the next section. Meanwhile we give some examples of operation transformations.

4.4.2 Example operation transformations.

For this set of example operation transformations we will consider those that might be written for the packed data representation. The structure transformation for that representation, introduced in 3.4.2, was as follows:

```
        type A = (s1: B; s2: C);
        type B = k..l;
        type C = m..n;          =  type A = 0.. (l-k+1)*(n-m+1)-1;
```

Firstly, consider a value-select operation on the old structure, whose implementation can be given as the following transformation:

```
        av's1    =>    av mod (l-k+1) +k
```

in which av stands for any data value of type A. On the left hand side of the transformation there is a simple value-select operation applicable to the old structure. On the right hand side is an arithmetic expression (mod being the integer infix operator that gives the remainder on dividing the first operand by the second), where av is now interpreted as being a value of the new structure, hence a value of an integer subrange. A similar implementation can be given for selecting the value of the second component of the old structure.

The next operation that can be implemented for this representation is construction of a new value of type A:

A#(bv,cv) => (cv-m)*(l-k+1) + (bv-k)

Given values bv and cv of types B and C respectively, constructing a Cartesian product value of the old type A can be implemented as evaluating the given arithmetic expression to give a value of the new type A.

If we now turn to implement a reference-select operation on the old structure, we find that it is not possible to return a reference to a component in the new structure since the 'structure' is a single integer value. However, it is possible to implement a compound operation which updates a component of the old structure:

ar.s1:= bv => ar:= (ar@ div (l-k+1))*(l-k+1) + (bv-k)

where ar is a reference to an item of type A, and bv is a value of type B. The left hand side is a compound operation consisting of a reference-select and an assignment operation, meaning that the value of the s1 component of the referenced data item is to be changed to bv. This is implemented as an assignment of a new value, computed from the current value of the referenced item and bv, to the referenced item. A similar implementation can be used for assigning a new value to the other component.

To summarise, therefore, in Figure 4.3 we give the complete set of structure and operation transformations for the packed data representation.

```
type A = (s1:B; s2:C);
type B = k..l;
type C = m..n;            =>   type A = 0..(l-k+1)*(n-m+1)-1;

av's1   =>  av mod (l-k+1) +k;
av's2   =>  av div (l-k+1) +m;
ar.s1:=bv  =>  ar:=(ar@ div (l-k+1))*(l-k+1) + (bv-k);
ar.s2:=cv  =>  ar:=(cv-m)*(l-k+1) + (ar@ mod (l-k+1));
A#(bv,cv)  =>  (cv-m)*(l-k+1) + (bv-k);
```

Figure 4.3 Transformations for packed representation

4.5 Applying Operation Transformations to Programs

In order to apply a data representation to a type in an abstract program, the two-stage process of matching and transforming applies to the operation transformations just as for the structure transformations discussed in section 3.5. First the representation must be found to match the type to be represented, and then, if the representation is chosen as the one to be implemented for the type, the program must be transformed to reflect the new representation.

4.5.1 Matching operations.

To match data operations it is necessary to determine that the representation implements each of the operations used in the program which is owned by the type to be represented. This may be done by making a scan of the program and, whenever an operation owned by the type is found, a search is made through the operation transformations of the representation to find a matching 'old' operation.

The identifiers in the operations of the representation must be interpreted according to the associations set up in the structure stage of matching. Any additional identifiers in the left hand side of an operation transformation are bound to specific program elements when a match is made, so that subsequently the right hand side identifiers can have appropriate substitutions made for them on implementing the operation transformation.

Example: Taking the packed data representation of 4.4.2, and applying it to the following types in a program:

```
  type PLAYING_CARD = (s:SUIT;r:RANK);
  type SUIT = 1..4;
  type RANK = 1..13;
```

we obtain the correspondences shown in section 3.5.1, including

Representation	A	s1
Program	PLAYING-CARD	s

On encountering the following operations in the program therefore

```
  p:= PLAYING-CARD#(q@'s,13);
```

where p and q are both variables of type PLAYING-CARD, there are two operations which the representation will match:

```
  A#(bv,cv)   will match   PLAYING-CARD#(...,...)
```

giving the additional identifier bindings

```
     bv  to  q@'s
     cv  to  13
```

and also

```
  av's1        will match   q@'s
```

giving the additional binding

```
     av  to  q@  .
```

Certain special operations owned by the type being represented need not appear in the operation transformations of the representation, but will not cause a match to fail. These operations are the ones that may apply to items of any type, no matter what structure the type may be specified to have, and will mean the same whether applied to an item of the type before or after the representation has been implemented. They are the assignment operation (:=) and the contents operation (@).

Example: In the previous example the assignment and contents operations in

```
p:=  PLAYING-CARD # (q@'s, 13);
```

belong to type PLAYING-CARD. However ,there is no need for them to be implemented by the packed data representatiòn since they will apply equally well to the type whether it is structured as a cartesian product or a single integer subrange.

If a representation requies these operations to be implemented in a special way, they can be included in the operation transformations along with the other implemented operations, and their defined implementation will be used rather than defaulting.

4.5.2 Transforming operations.

On deciding that a representation which has been matched to a data type in a program will be implemented for that type, the program is transformed accordingly. In the case of the operation transformations, each operation in the program which has been matched with the left hand side of an operation transformation is replaced by the right hand side of the transformation. All identifiers in the right hand side which have had correspondences established with program values in the matching stages are replaced accordingly.

Example: For the two operations considered in the previous subsection, their right hand sides in the representation description are respectively (see 4.4.2):

```
      (cv-m)*(l-k+1) + (bv-k)
 and
      av mod (l-k+1) + k
```

On substituting the identifiers bound by the structure matching and each operat matching these become:

```
      (13-1) * (4-1+1) + (q@'s - 1)
and    q@ mod (4-1+1)  + 1
```

In the first of these, the operation q@'s, originating from the program before application of the representation itself, uses the implementation given in the second line. Thus the original program statement

```
      p:= PLAYING-CARD#(q@'s, 13);
```

is transformed into

```
      p:= (13-1)*(4-1+1)+ (q@ mod (4-1+1)+1 - 1);
```

4.5.3 Optimisation.

Of course applying a set of separate transformations to a program may lead to a new program in which various redundant expressions, or expressions which may be simplified, will occur. Various degrees of optimisation may therefore be applied after a program transformation. In the experimental system a simple optimisation of evaluating all arithmetic expressions where possible was applied and found to be largely sufficient.

Example: The above transformed program statement would be simplified by the experimental system to

```
      p := 48 + q  mod 4;
```

The transformational approach does not necessarily imply a macro-like implementation of all data operations. If an operation implementation was a non-trivial piece of program and it was required several times in a program, implementing it as a single procedure declaration with a procedure call at each point of use might be preferable. However, the additional complications introduced when procedures are used have not been investigated (see section 6.2).

CHAPTER 5

ASSISTING THE SELECTION OF REPRESENTATIONS

5.0 Summary

Having described an approach to data representation by program transformation, and investigated some of the notational techniques such an approach could adopt, we next go on to see how using the approach could assist the programmer in the selection of data representations.

First we give a brief description of the experimental system that was used as a test-bed during the investigations. Next, the matching of representations is further discussed to show how representation selection may be aided by more tailored matching.

The choice of representations is aimed at obtaining a suitably efficient final program, in terms of storage space and execution time. The evaluation of representations in terms of their effect on program costs is discussed in the third section. Finally, by use of examples from the experimental system, the way in which the application of a series of transformations can be guided to produce an acceptably efficient target program is considered.

5.1 The Experimental System

As a means of gaining experience with the problems involved in taking a transformational approach to data representation, an experimental interactive system was programmed to run on the IBM 370/168 at Newcastle University. The objective in implementing the system was to provide a test-bed for examining the selection of data representations, rather than to investigate the detailed design of such a system itself. We shall therefore give an overall view of the system in order that the results of the experiments carried out in using it may be understood, but avoid a description of its detailed implementation.

5.1.1 Overall form of the system.

An overall diagram of the system is shown in Figure 5.1

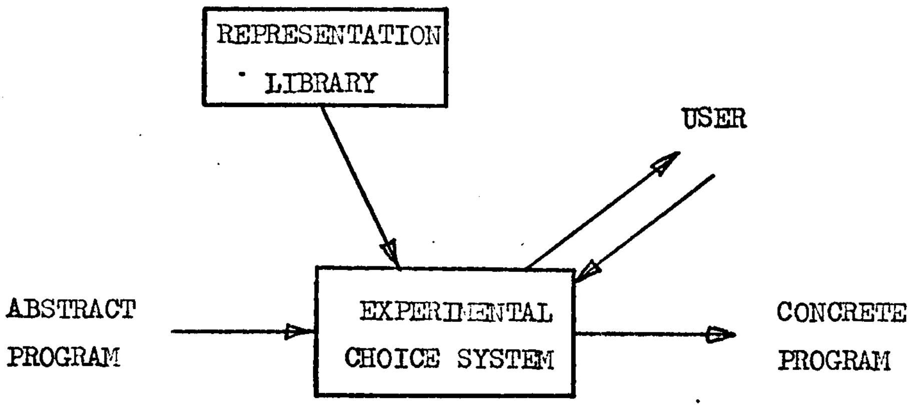


Figure 5.1 Overall System Diagram

The representation library consists of a file of all the data representations available for selection and implementation in the user's abstract program. The library is searched for suitable representations whenever a specified type in the user's program is to have a representation chosen for it. Initially, the user's abstract program is input to the system. Then, under the user's control, the system helps in the selection, evaluation and implementation of representations for the data types in the abstract program, until finally all the types and structures in the program are basic to the target language, and the final concrete version of the program can be output. The user controls the representational choice process by interactively monitoring the order in which data types are represented, and by choosing the representation actually to be implemented from the set of feasible representations selected and evaluated by the system for any given data type.

The notation used to express both the representations stored in the library and also the abstract programs has been described already. A part of the system, that will not be elaborated further here, act as a syntax analyser and checker for the notation, and generates an internal code version of either data representations or abstract programs. This internal code, which has been checked as forming a syntactically valid representation description or program, is the form in which representations are stored in the library, and programs input to the choice system, respectively.

Taking a closer look now at the internal organisation of the experimental system, it consists of several logical sections indicated in Figure 5.2. We shall consider the purpose of each of these sections in turn.

The abstract program, on being initially input to the system, is stored there in its internal code form. Whenever the set of suitable representations are to be examined for a given data type in the program , the current stored program is accessed to determine the structure specification and the data operations of the given type. When a chosen

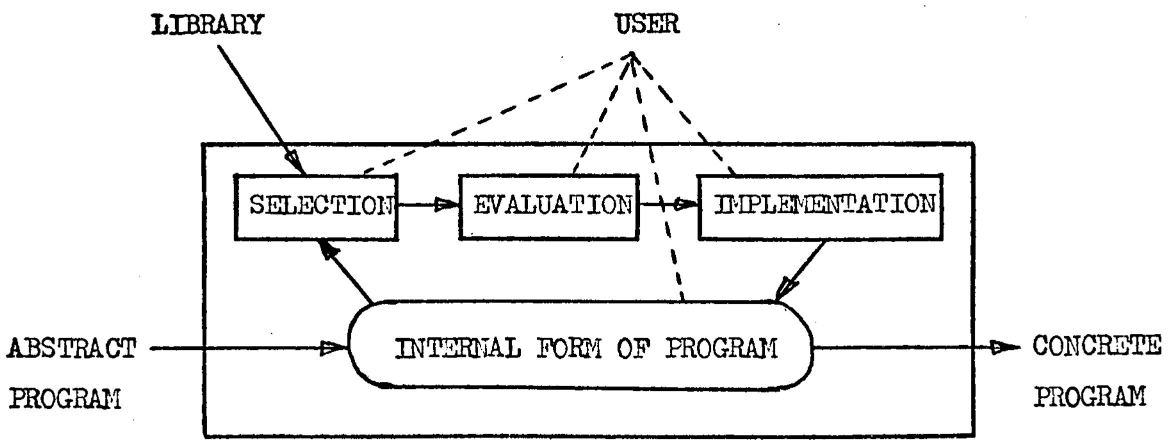


Figure 5.2   Internal Organisation of System

representation for a data type in the program is implemented, the stored program is transformed to incorporate the new representation, both in the structure specification of the type and in the operations performed on items of that type. In other words, the current program stored in the system always reflects the set of data representations that have been implemented up to that point in the representational choice process. Once a chosen representation has been implemented, the stored form of the program hss been irrevocably changed, and it is not possible to subsequently restore it by 'undoing' the representation of a given type, since later representations may have in their turn transformed the whole of or parts of the structure specification or operations of the type.

5.1.2 Use of the system.

At any given stage in the representation process, the system user has the choice from the current data types in the program of which type to investigate next. By issuing an appropriate command to the system he may request the system to search its library for all the suitable representations for a given type. The selection of the set of feasible representations for that type involves matching the current specification of the type in the program with each representation in the library. An evaluation of each of the feasible representations is made in order to estimate the storage cost of each representation. This evaluation, and the evaluation of execution time cost, is dealt with further in section 5.3. Having seen the choices of representation available to him, and been given some guidance as to the relative costs of each choice, the user is then free to select one of those choices to be implemented. The way in which implementation of a representation transforms the stored program has been described already. If the user decides that none of the possible choices is suitable at that point in the representational choice process, he is free to specify a different abstract type and investigate its representation.

A large amount of guidance is therefore required from the user of the experimental system, especially in choosing the order in which the abstract types in the program are to be represented. This freedom was designed into the system on purpose so that different orders of choice could be easily tried and the consequences of different selection strategies investigated. The interactions between the choices of representations, particularly between hierarchically related types such as the representation of a set and the representation of its elements, can become complex. Some of the possible options available to the user are discussed in section 5.4.

When all the types and structures have been represented in terms of basic target language constructs, the final program can be output by converting the internal form into the syntax of the target language (though this was not a feature included in the experimental system).

5.2. Matching Representations

In order to determine the feasible representations that may be used for a given data type at some point in the representational choice process, a scan of the library of representations is made in order to find those which match the type specification. In fact, in the experimental system, matching representations is a three-stage process, as shown in Figure 5.3. Firstly, the structure specification of the type being represented, and any of its constituent types if necessary, is matched with the 'old' structure specification of the representation (as described in subsection 3.5.1). If it is found to match, a conditional expression in the representation description, if present, is evaluated. This expression allows the representation writer to describe the circumstances in which the representation may be chosen. This condition is described more fully in subsection 5.2.1. If the condition is true, the third stage is undertaken, consisting of a scan of the procedural part of the program checking that each data operation owned by the data type being represented is implementable by the representation (this was described in subsection 4.5.1). Only if all three possible stages in the matching process are successful can that particular representation be considered as a feasible one for the given data type.

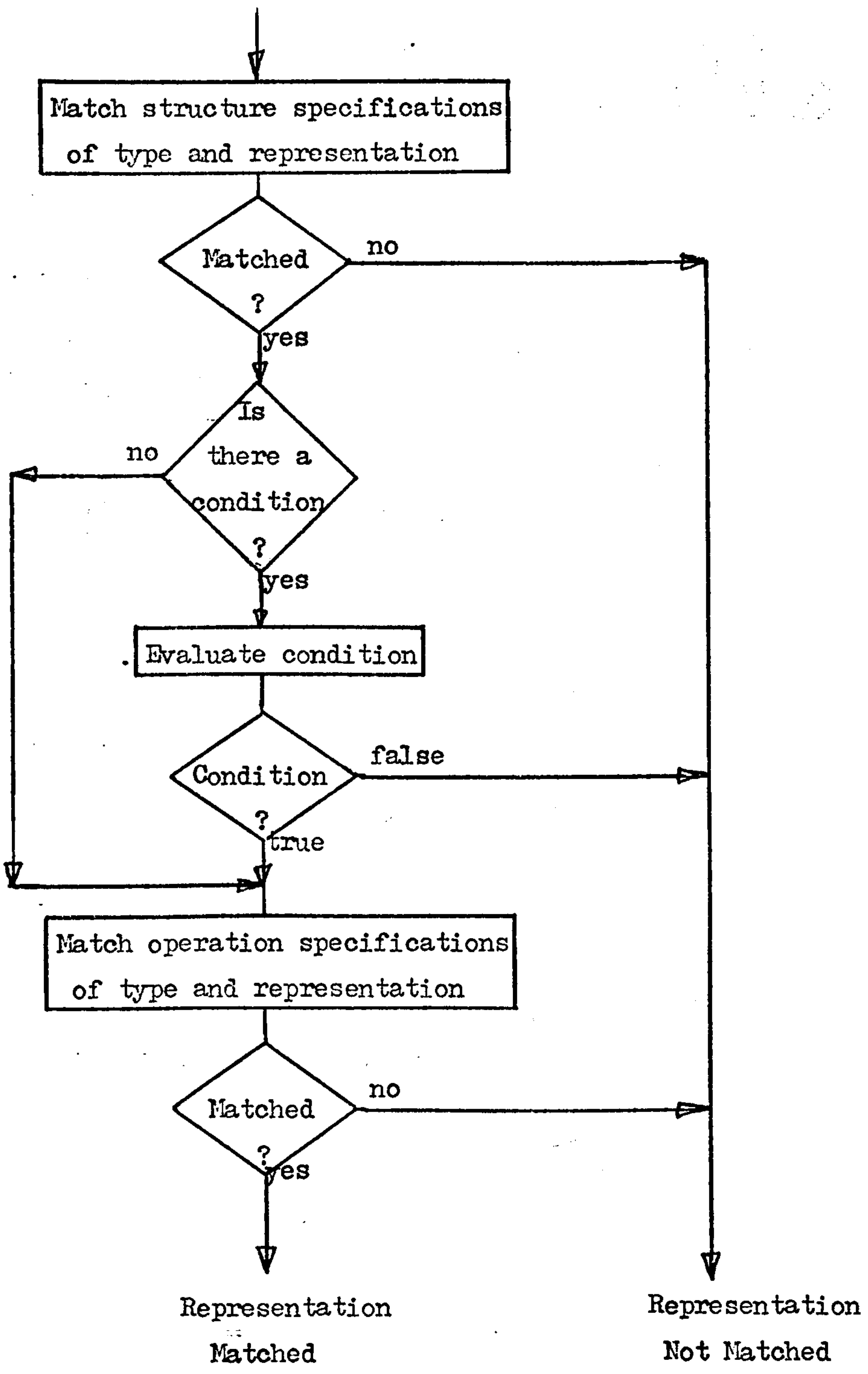

Match structure specifications of type and representation
Matched ?
no
yes
Is there a condition ?
no
yes
Evaluate condition
Condition ?
false
true
Match operation specifications of type and representation
Matched ?
no
yes
Representation Matched
Representation Not Matched

Figure 5.3 Matching Representations

5.2.1 Conditions to aid selection.

For some representations, the conditions under which a representation is applicable to a given data structure cannot be determined by the structure and operation matching alone. For this reason, a representation may include a conditional expression, defined by the writer of the representation, which is evaluated during matching. The expression may involve values, such as subrange bounds and structure properties, which are determined during the structure matching of the representation to a given type. Only if the condition evaluates to 'true' may the may the matching continue to determine whether the representation is applicable to the type.

The condition may be used in two distinct ways, both of which were found useful in the experimental system. The first use is in order to specify a condition which must be true for the representation to be applicable.

Example: The following representation includes such a condition:

```
type A = (bb:B|cc:C); type B = k..l; type C = m..n;
     => type A = k..n;
condition l<m;
```

In this case, it is possible to represent the discriminated union of two subrange values by a single subrange value having the same total range, provided the two initial subranges are disjoint. The condition, expressed in terms of the subrange limits, which will be bound to specific values when matched to the structure of a given data type, is necessary to ensure the representation can only be fully matched if the ranges do not overlap.

The second use for the condition is pragmatic, in that many representations may be logically applicable to a given structure, but it would not be possible to consider them as feasible in practice because of other constraints. The writer of a representation may therefore use the condition to reduce the number of feasible representations considered for any given structure by stating the constraints he wishes to impose on the matching of the representation.

Example: Take the packed data representation of a cartesian product of two subranges:

```
type A=(bb:B; cc:C); type B = k..l; type C = m..n;
    => type A = 0..CARD(A)-1;
condition CARD(A)<CARD(int);
```

Here we have used one of the type property functions CARD, which gives the cardinality of the argument type. (See Appendix I for a description of the properties catered for in the experimental system). In this example CARD(A) is equivalent to (l-k+1)*(n-m+1).

The condition will only allow the representation to be applied in this case if the cardinality of the type being represented is less than that of the basic type int. The reason for this restriction is pragmatic in that, though the representation can in theory be applied to such a type no matter what its cardinality, in practice the resultant subrange must itself be represented eventually. If its cardinality is greater than that of an int value, it cannot be represented in terms of a single int, and therefore it must be split into manageable subranges again. Rather than allow the initial subranges to be combined only to be later split again, the writer of the representation has excluded its application in this case.

5.2.2 Example of matching.

The matching process will be illustrated in full for an example to bring together the structure matching (3.5.1), operation matching (4.5.1) and use of the conditional expression (5.2.1).

Take as an example the following representation, written in a form similar to that in which it would be added to the representation library of the experimental system.

```
rep subdu2;
    type A = (bb:B|cc:C);
    type B = k..l;
    type C = m..n;
        => type A=0..CARD(B)+CARD(C)-1;
           type B=k..l; type C= m..n;
    condition CARD(A)<CARD(int);
    operations
       ax'bb       =>  ax+k;
       ax'cc       =>  ax-CARD(B)+m;
       ax?bb       =>  ax<CARD(B);
       ax?cc       =>  ax>=CARD(B);
       A#bb(bx)    =>  bx-k;
       A#cc(cx)    =>  cx+CARD(B)-m;
endrep
```

(The syntactic differences required to enable this representation to be added to the experimental library are minor ones which identify ax as a

value of type A, and bx and cx as values of type B and C respectively. They have been omitted here for the sake of clarity).
The representation, named 'subdu2' , allows a discriminated union of two subranges to be represented by a single subrange. Values of type B, discriminated by the selector tag 'bb', will be represented by a value in the range 0..CARD(B)-1, while the other alternative, with selector tag 'cc', will be represented by values in the range CARD(B)..CARD(B)+CARD(C)-1. The operations in the representation allow the value of each alternative to be obtained (these value-select operations presume that the programmer has already determined that the appropriate alternate holds for the operation to be valid), allow it to be determined whether a given alternate is currently assigned, and allow the construction of a value of one alternate or the other. The specification of types B and C are repeated on the right hand side of the structure transformation since these types keep the same specification after the representation has been implemented, and values of the types are used on the right hand sides of the operation transformations.

Now, suppose we wish to select a suitable representation for type T specified in a program as:

```
type T = (sel1:U|sel2:V);
type U = 1..10;
type V = 3..7;
```

Then upon encountering the above representation in the scan of the library, a match will be attempted. The first stage of the matching process will succeed in matching the structure specifications, and will have the additional effect of setting up associations between identifiers in the representation and the actual type specifications as follows:

Representation identifier	Program identifier
A	T
bb	sel1
B	U
cc	sel2
C	V
k	1
l	10
m	3
n	7

With these associations set, the conditional expression in the representation may be evaluated as

CARD(T) < CARD (<u>int</u>)

which would be true in this case, since CARD(T)=15, which is less than the cardinality of the basic integer type.

The structure matching and condition evaluation having both been successful, the next stage involves matching each operation upon type T in the program with the operations implemented by the representation. For example, if the following statement occurred in the program:

```
        if t?sel1  then x:= t'sel1;
```

(where t is a variable of type T and x is a variable of type <u>int</u>). there are two operations owned by type T, t?sel1 and t'sel1. In the list of operations in the representation we have the following two definitions:

```
        ax?bb      =>    ax < CARD(B);
        ax'bb      =>    ax+ k:
```

Using the associations already established, the first matches t?sel1, giving the additional association of ax with t. This association will be used upon implementing the operation in order to enable its implementation to be written as

$$t<10 \qquad \text{(note CARD(U)=10)}$$

A similar match is possible for the other operation, so that the original statement would therefore be transformed into the following upon implementation of the representation:

```
if t<10 then x:= t+1;
```

5.3 Evaluating representations

In this section we consider the evaluation of feasible data representations, and present the simple evaluation scheme that was used in the experimental system as a means of giving insight into the problems involved.

5.3.1 Aims of evaluation

Given an initial abstract program, we wish to choose suitable representations for the data used in the program, in order to arrive at an efficiently executable target language program. In other words, we wish to reduce to an acceptable level the cost of execution of the resultant program, both in terms of the storage used and the time taken, and therefore require some means of evaluating this cost. One way of evaluating the resultant program would of course be to monitor its actual execution on some suitable test data. This method has the disadvantage of being slow and wasteful in this context, however, requiring the time and expense of compilation and execution of the program as well as the selection of one, or possibly several characteristic sets of test data.

Complete accuracy of cost evaluation is not generally required when choosing data representations. Rather, good estimates of costs are sufficient to compare different representations, especially when the costs being compared may differ by an order of magnitude. A static evaluation of a program, estimating its execution costs from the static program text, therefore appears preferable in the circumstances. For a given target language, the average storage space and execution time required for each of the basic data types and operations in the language may generally be determined. If, in addition, information is available about the relative frequency of execution of the alternative and repetetive constructs used in a given program (for example, the probability that the conditional in an <u>if</u> or <u>while</u> statement will evaluate to <u>true</u>), total estimated space and time requirements can be calculated from the program text.

When choosing data representations using the transformational approach described here, however, a sequence of representational choices will have to be made for the various abstract data types before a target language program is achieved. If costs estimates can only be made on target language programs, comparisons will only be possible between complete sequences of representational transformations. This will provide little help in making a choice for an individual representation, and therefore be less effective in guiding the sequence of chosen representations to a resultant efficient program. Exhaustive evaluation of all of those possible sequences of representational choices that result in a target language program is infeasible because of the combinatorially large number of possible sequences.

For these reasons, an attempt was made to see if useful estimates of cost could be derived in order to compare the alternatives for a single representational choice. In the case that an alternative results in an implementation involving only types and operations of the target language, the cost estimates would reflect the real target language costs. If, on the other hand, the alternative was expressed in terms of further abstract types and operations, not directly implenented in the target language and therefore requiring further representational choices to be made, estimates would be made of the costs of the abstract types and operations. The latter estimates must inevitably be only approximate, since the true costs will depend on the subsequent representations chosen for the abstract entities involved. One of the points of investigation concerning the evaluation procedures used in the experimental system was whether such estimates could still be a useful aid in selecting suitable data representations.

5.3.2 Related work on evaluation

Apart from the mathematical analysis of algorithms (see, for example, Knuth [31]), which provides a mathematical approach to estimating program execution times, and is at present beyond the scope of automation, various other approaches to program evaluation using the static program text have been attempted.

Wegbreit [53] describes a system which is able to perform completely automatic analysis of simple Lisp programs, with no additional information provided by the programmer beyond the program text itself. He shows how a closed-form expression for program running time can be obtained, expressed in terms of the size of the input. However, in its described state his system is only suitable for analysing simple Lisp programs.

Cohen and Zuckerman [9] present a means of estimating program efficiency in which the program to be analysed, written in a special language, is first transformed into a symbolic formula representing the execution time of the program. This formula is then passed to an interactive system, where the user may bind symbolic or numeric values to the variables in the formula (including such variables as the basic operation execution times and the probabilities of conditional expressions evaluating to true or false), simplify the formulae, and, when sufficient variables have been found, to plot the numeric results for the program execution times.

Middleton [43] models a program as a graph, and shows how storage space and execution time requirements for a given program can be derived from the resource equations applicable to each of the basic graph constructs. For example, for a FORTRAN Do-loop construct he gives resource equations of:

$$\mathrm{TIME}\ (\mathrm{DO}(N,a)) = T_0+N*(T_1+\mathrm{TIME}(a)+T_2)+T_1$$
$$\mathrm{STORE}\ (\mathrm{DO}(N,a)) = S_0+S_1+S_2+\mathrm{STORE}(a)$$

where N is the number of complete cycles of the loop
a is the action forming the body of the loop
T_0, T_1, T_2 and S_0, S_1, S_2 are the time and storage requirements for initialisation, testing and incrementing the control variable respectively.

This model allows the effectiveness of certain time/storage trade-offs to be analysed, which is one of the factors that is of importance in comparing data representations.

More specifically dealing with data representations, Tompa [51] has described an evaluation method involving an evaluation matrix. By using a special language ('Counter') devised for the purpose, first introduced in an earlier paper [22], the cost of each storage schema (i.e. representation) from which a choice can be made is expressed in terms of parameterised formulae. These are then used to construct an evaluation matfix in which the element pair in row i and column j represents the run time and storage space contributed by the jth structure when implemented using the ith representation. Tompa shows [51] that in general, however, the least overall cost (expressed as a space-time product) is not necessarily obtained by selecting the least-cost representation for each individual structure. To avoid the exhaustive examination of all possible combinations of structures and representations that would otherwise be necessary, he presents a branch and bound algorithm to reduce the search space to a manageable size.

Using the same approach as Tompa, Low [39,40] also evaluates data representations by using space and time cost functions associated with each representation. The cost formulae, parameterised by information such as the maximum number of components of the structure being represented, are determined manually from the primitive storage required and the execution time taken by implementing code for the representation. Information about the use of the structures to be represented is derived from the program in which they are used in order to provide the appropriate parameters for the evaluation. This information is derived partly from monitoring sample executions of the program, and partly by direct interrogation of the user.

Both Tompa and Low consider the representation of an abstract data structure directly in terms of a concrete representation. They are therefore able to estimate the cost of using a particular representation directly in terms of the cost of implementing primitives.

In the approach described here, however, several representational transformations will in general be made before a given abstract structure is implemented in terms of concrete primitives. Because of this, the approach to evaluating representations was made more difficult than in the work described in this section.

5.3.3 Approach used in the experimental system.

A static evaluation approach was investigated in the experimental system, estimating the program execution costs from the static program text. To determine the cost of using a particular data representation for a given type in a program, the representation was implemented and the resultant transformed program evaluated. By comparing the costs of the various resultant programs, therefore, the effectiveness of each of a set of feasible representations for a given type can be compared.

A program is evaluated in terms of the storage space required to hold all the data items used by the program, and the time taken to execute the complete program; no attempt is made to give a single overall cost, say in terms of a space-time product, since the relative importance of the two factors may vary according to particular circumstances, often the time being minimised within a fixed maximum space.

When static program analysis is carried out, certain information concerning the behaviour of the program at execution time, which cannot be derived from the program text alone, must be additionally supplied. In order to calculate storage requirements it is necessary to specify, for example, the maximum expected number of components in a sequence, or the maximum expected number of non-default valued range elements in a sparse array. Similarly, to calculate execution times it is necessary to specify, for example, the expected number of time a while loop will be executed, or the expected probability that the condition in an if statement will evaluate to true. One way to obtain such information is to monitor the actual execution of the algorithm being programmed, as suggested by Low [39]. Any implementation may be used for the algorithm in order to do this monitoring, it is not necessary to choose an efficient implementation, since this of course is the object of the evaluation in the first place. Having obtained the usage information, this can then be used to select an efficient implementation of the algorithm. If the programmer has no idea of how his algorithm will behave, it may be necessary to perform this initial trial and monitoring process. However, the programmer will often be able to give reasonable estimates of the performance of his algorithm in order to supply the usage information in his abstract program. (Indeed, it may be suggested that the programmer who cannot estimate such information does not know sufficient about his algorithm in the first place). If the initial usage estimates are found to be inaccurate in the light of the actual program execution, they can be revised and the representational choice system can be used to determine whether a different implementation would be more suitable.

In the experimental system, the programmer must provide all the necessary usage information in the abstract program initially input to the system. In a more sophisticated system, it might be possible to derive some of the information from careful analysis of the program. For example, if the conditional in an if statement consisted of a predicate testing whether a particular data item was an element of a given set, the probability that the conditional would evaluate to true could be determined from the ratio of the average number of elements in the set to the total possible number. In fact, when writing representations for the library this kind of reasoning is needed to specify such information in the implementations of the data operations.

In the experimental system, such probabilities etc. must be specified as formulae by the writer of the representation. In a more sophisticated system, they might be derived automatically. Such calculations for complex algorithms, however, may require techniques such as the mathematical analysis mentioned earlier [31], and so be beyond the present scope of automation.

A refinement of the requirement for full information on program usage to be supplied by the programmer, would allow for interactive use, in which the system requests from the programmer only those items of information needed at any given moment in the choice process.

In the following three subsections, further details are given of the storage space and execution time evaluation methods used in the experimental system, and some comments on estimating costs for abstract programs are made.

5.3.4 Storage space evaluation.

In order to simplify the evaluation of storage requirements and unify the data items in a program, the experimental system uses a special type 'GLOBAL' to encompass all the program variables. The storage used by a program therefore consists of a single instance of this type.

Example: Given an initial program with the following type and variable declarations:

```
type A = - - -;
type B = - - -;
type C = - - -;
var a:A; var b:B; var c:C;
```

this is held in the experimental system in the form:

```
type GLOBAL = (a:A; b:B; c:C);
type A = - - -;
type B = - - -;
type C = - - -;
var g:GLOBAL;
```

and all references to the variables a,b and c are instead held as reference-selections g.a, g.b and g.c from the single global variable g.

Introducing such a global type also enables representations to be chosen for this type, so that, for example, all the data in the program can be represented in a single array of integers (corresponding to the basic storage of an assembler-oriented target machine). Evaluating the storage required by a program therefore consists of working out the storage required by an instance of type 'GLOBAL'.

The cardinality of a data type is the number of distinct values that belong to the type, and which can therefore be assigned to an instance of the type. If c is the cardinality of a type, an instance of that type will require a minimum of

$$\lceil \log_2 c \rceil$$

bits of storage to hold any value of the type. This is therefore the minimum amount of storage that a representation of that type could use, though it may use more storage. Each kind of structuring combinator has an associated method of calculating the cardinality CARD(T) of a type T specified in terms of that combinator:

Integer Subrange:	<u>type</u> T = m..n
	CARD(T) = n-m+1
Cartesian Product:	<u>type</u> T = (s1:T1; s2:T2; ... ; Sn:Tn)
	CARD(T) = CARD(T1)xCARD(T2)x ...xCARD(Tn)
Discriminated Union:	<u>type</u> T = (k1:T1|k2:T2| ... |kn:Tn)
	CARD(T) = CARD(T1)+ CARD(T2)+...+CARD(Tn)
Array (Non-Sparse):	<u>type</u> T = <u>array</u> T1 <u>of</u> T2
	CARD(T) = $\text{CARD(T2)}^{\text{CARD(T1)}}$
Powerset(Non-Sparse):	<u>type</u> T = <u>set of</u> T1
	CARD (T) = $2^{\text{CARD(T1)}}$

The cardinality of a basic structure type such as <u>int</u> depends upon the definition of the type as implemented in the target language.

In the cases of sparse arrays (in which most domain values map onto the same default range value), sparse powersets (in which each set value contains only a few of the possible set element values) and sequences (which would otherwise have infinite cardinality), the calculation of the cardinality is somewhat more complex and depends on the maximum number of components the structures will contain.

Sparse Array: <u>type</u> T = <u>array</u> U <u>of</u> V (MAXCOMP=m)

Here, m denotes the maximum number of domain values, of type U, that will map onto non-default range values, of type V.

Let t_m = CARD(T) with MAXCOMP=m, u = CARD(U), v = CARD(V).

Then $t_0 = v$ (a single default value).

$t_i = t_{i-1} + v \binom{u}{i} (v-1)^i$ $(i>0)$

(i.e. increasing i by one increases the cardinality of the array by the number of possible non-default domain-range combinations with exactly i non-default entries).

Thus $t_m = v \sum_{i=0}^{m} \binom{u}{i} (v-1)^i \simeq \frac{u^m v^{m+1}}{m!}$ for $u \gg m$.

Sparse Powerset: <u>type</u> T = <u>set</u> <u>of</u> U (MAXCOMP=m)

Here, m denotes the maximum number of set elements, of type U, in any value of type T.

Let t_m = CARD(T) with MAXCOMP=m, u=CARD(U).

Then $t_0 = 1$ (the empty set)

$t_i = t_{i-1} + \binom{u}{i}$ (i>0)

(i.e. increasing i by one increases the cardinality of the set by the number of set values with exactly i elements).

Thus $t_m = \sum_{i=0}^{m} \binom{u}{i}$

$\simeq \frac{u^m}{m!}$ for u>>m.

Sequence: <u>type</u> T = <u>sequence</u> U (MAXCOMP=m)

Here, m denotes the maximum number of sequence elements, of type U, in any value of type T.

Let t_m = CARD(T) with MAXCOMP=m, u = CARD(U).

Then $t_0 = 1$ (the empty sequence)

$t_i = t_{i-1} + u^i$ (i>0)

(i.e. increasing i by one increases the cardinality of the sequence by the number of sequence values with exactly i elements).

Thus $t_m = \sum_{i=0}^{m} u^i$

$$= \begin{cases} \frac{u^{m+1}-1}{u-1} & u>1 \\ m + 1 & u=1 \end{cases}$$

Given a set of abstract type specifications in a program, the above formulae enable the minimum theoretical storage space required for the program to be calculated. In practice, the representations chosen for the abstract types in the program will result in more storage space being used, because to compress the data to its most compact form would require unacceptable overheads for packing and unpacking individual data items as they were needed for manipulation in the program. For example, often an integer subrange of small cardinality, such as <u>type</u> T = 1..10
will be represented as an <u>int</u>, with a resultant increase in the storage evaluation, simply because, if the instances of the type are frequently manipulated in the program, no extra execution time overhead need be incurred for packing and unpacking the values into a smaller number of bits.

5.3.5 Execution Time Evaluation.

The calculation of the estimated execution time of a given program can be broken down into two parts. Firstly the determination of the time taken to perform each individual data operation in the program, such as selecting a component of a structure, constructing a new value from some component values, or assigning a new value to a named data item. Secondly, the combination of the individual operation times according to the expected flow of control in the program to produce an overall time for execution of the complete program. For data types and structures which are basic to a given target language, the average execution times for the data operations on those types and structures may be determined for the particular implementation of that language that will be used to compile and execute the final program. For example, Wichmann[56] has produced such analyses for several different implementations of Algol 60. On the other hand, estimating the time to execute data operations for abstract types, which have not yet been represented in target language terms, is very difficult. It is considered further in the next subsection.

Meanwhile, assuming times have been allocated for each individual data operation, it remains to combine these times according to the structure of the program:-

Compound Operation: For a compound data operation, such as

ax.curr := ax'aa'index(ax'curr)'next;

the execution time is the sum of the times for the component operations.

Compound Statement: S ≡ <u>begin</u> S1; S2; ...; Sn <u>end</u>

time (S) = time (S1)+time(S2)+ ... +time(Sn)

If Statement: The probability p that the condition C of the <u>if</u> statement will evaluate as <u>true</u> at execution time is provided by the programmer (syntactically in the experimental system it is included as an expression enclosed in '%' brackets after the condition).

S ≡ <u>if</u> C %p% <u>then</u> S1 <u>else</u> S2

time (S) = time(C) + pxtime(S1) + (1-p)xtime(S2)

or alternatively if the '<u>else</u>' part is omitted

S ≡ <u>if</u> C %p% <u>then</u> S

time (S) = time (C) + p x time(S1)

While Statement: The average number of times f that the body of the loop will be executed is provided by the programmer.

S ≡ <u>while</u> C %f% <u>do</u> S1

time (S) = (f+1) x time (C) + fx time (S1)

For Statements: As in the <u>while</u> statements, the average number of times the loop will be executed is provided by the programmer.

The first form of <u>for</u> statement iterates over (at most) all the values that can be taken by a named data item N of a type T, the owner of the statement.

S ≡ <u>for</u> N %f% <u>do</u> S_1

time (S) = time (N) + fxtime(S_1)

If no exits are made to terminate the iteration early,

f = CARD (T).

The second form of for statement iterates a statement for each of the element values in a given set or sequence value V of type T.

S ≡ for X in V %f% do S1

time (S) = time (V) + fxtime(S1)

If no exits from the loop are made, the number of iterations will depend on the average number of components in the set or sequence, a number which can be specified by the programmer when declaring the set or sequence type,

f = AVCOMP (T).

The third form of for statement iterates for a named data item N taking values over a given integer subrange, the bounds of which are specified by integer values V1 and V2

S ≡ for N:V1..V2 %f% do S

time (S) = time(N)+time(V1)+time(V2) + fxtime(S1)

If no exits from the loop are made, and V1, V2 are constants that are known at the time of evaluation,

f = V2 - V1 + 1.

In the experimental system no extra times were added in the above calculations to allow for branching or looping overheads. In general, such overheads will be small compared to the total time of execution for the statements with which they are associated. If necessary, for greater accuracy, the overheads incurred by such branching etc. could be determined for the specific target language being used, and the appropriate additional amounts included in the above formulae.

5.3.6 Problems of evaluation to assist selection.

For a given type, specified by a particular data structure, it has been shown in subsection 5.3.4 how the minimum storage space required for values of such a type may be determined. This minimum conceptual storage may be used as a representation-independent way of assessing storage cost. Unfortunately, there is no equivalent way of assessing the cost of operations on structures in a representation-independent fashion. The cost of inserting an element into a set, for example, depends entirely on the representation of the set, and in turn on the costs of the operations in that representation that implement the insertion. The true cost of such an insert operation will only be known when the set has been fully implemented in terms of primitive language operations.

In the experimental system, the effects of providing very crude estimates of the costs of abstract operations was investigated. However, these experiments did not provide a useful guide to the ultimate execution time costs of the chosen representations. In general different representations may have different balances of costs among the operations they implement. Whereas with storage it is possible to set a minimum on the amount of storage that may be used to represent a data type instance, with data operations no such minimum can be set. Representations can be devised that minimise the cost of one of their operations at the expense of making the other operations more time consuming. However, it would not be possible to assume such a minimum for each of the operations of a representation, since they cannot simultaneously achieve these minima.

Examples of the storage evaluation provided by the experimental system during the choice process are shown in the next section. They demonstrate that the storage evaluation alone is only of small use in guiding the selection of representations.

5.3.7 Evaluation and the transformational approach.

Despite the inadequate evaluation of representations that could be provided during the choice process, the evaluation mechanisms incorporated in the experimental system are still of importance in allowing a static program evaluation to be made once a target language form of the program has been reached. The evaluation experiments that were carried out did show that information on control flow, which is required in making an evaluation of execution time, could be carried through the representational stages from abstract to concrete program. It is reasonable to ask the programmer to supply such information for the abstract program that he has written. However, it would be foolish to expect him to give such information for the concrete form of the program after it has undergone several transformations. If it were not possible to carry the information through from abstract to concrete program, evaluating representations using the transformational approach would be extremely difficult from the program text, and would require dynamic execution of the concrete program to gather execution-time statistics. Our experiments have demonstrated that the information can be carried through, and that static evaluation is therefore feasible.

The means by which the selection of representations during the choice process can be guided by evaluation is still largely unresolved. One possibility is to use the storage evaluation as a coarse filter to eliminate representations which require excessive storage and so reduce the space of feasible representations. If the search space can be reduced to a relatively small number of sequences of transformations applied to the program, static evaluation of the concrete program resulting from each sequence can be used to select the most efficient. The advantage of using a system of the kind presented here is that the investigation of different sequences of transformations is made relatively easy. With the implementation of transformations, the maintenance of evaluation information, and the final evaluation of concrete program forms carried out automatically by the system, the user of the system can afford to investigate alternative sequences of representational choices.

5.4 Directing the Application of Transformations.

In order to see how the transformational approach may be guided by a programmer in conjunction with machine support, we next consider an example of use of the experimental system. The example is used to demonstrate the storage evaluation included in the system, and in particular to show how the user is able to direct the application of transformations in order to achieve a final target language program.

The example demonstrates also some of the problems and some of the expediences taken to avoid problems in the system. The amount of guidance given directly by the system in choosing representations is shown to be small, and possible means for assisting the programmer further in the choice of representations are considered.

5.4.1 An example of use of the experimental system.

This extended example will show the complete representation of the data in a program, starting with the programmer-written abstract form and finishing with a target language compatible form. The example to be used is the Birthdays Problem, described initially in 1.2.1 and used for various other examples in the rest of the text.

An abstract program to solve the problem was given in 4.3.4 and the actual program input to the experimental system at the start of the session is shown in Appendix II. The latter differs from the former only in including some additional information. For two of the data type declarations:

```
    type TABLE = array DAY of GROUP  (AVCOMP =50, MAXCOMP =100);
    type GROUP = set of PERSON       (AVCOMP = 1, MAXCOMP =10);
```

the additional information is given by the programmer to initialise the properties of the relevant types concerned with program dependent usage of the types. In each case the information specifies the maximum and average number of components that an instance of the type will contain. In the case of the array, it refers to the maximum and average number of array elements that differ from a common value (in this case the empty GROUP set). In the case of the set, it refers to the maximum and average number of elements that a set will have. The information is used to calculate the amount of storage required by the data, as discussed in 5.3.4. The other difference is that in the body of the abstract program the programmer has provided additional information about the frequency of execution of the loops, which is required for carrying out the execution time evaluation discussed in 5.3.5.

We shall presume for this example that the target language for which the user wishes to represent the data is one which includes only variables of basic type int, and also singly dimensioned arrays of elements of type int. In other words, at the end of the representational choice session the data (i.e. the single instance of type GLOBAL) should have the form of a Cartesian product whose elements are either of type int, or of types specified as arrays of int with integer subrange index types.

We now give a commentary on the interaction of the user with the system when selecting and implementing representations for the data.

User input to the system is distinguished by the prefix ">". Once the initial abstract program has been input to the system, the system is ready for use:

```
** REPRESENTATIONAL CHOICE SYSTEM   **
COMMANDS:
   TYPES, REPS <TYPE>, EVAL, PRINT, QUIT,
   SPLIT <TYPE> <SEL> <SEL>
>PRINT
TYPE GLOBAL=(P:PERSON;D:DAY;T:TABLE);
TYPE TABLE=ARRAY DAY OF GROUP  (MAXCOMP=100 AVCOMP=50 );
TYPE DAY=1..366;
TYPE GROUP=SET OF PERSON  (MAXCOMP=10 AVCOMP=1 );
TYPE PERSON=1..5000;
VAR G:GLOBAL;
  BEGIN
  G.T:=(TABLE#ALL(GROUP#EMPTY));
  READ(G.P);
  WHILE G@'P¬=0 DO
    BEGIN
    READ(G.D);
    G.T.INDEX(G@'D):INSERT(G@'P);
    READ(G.P);
    END;
  FOR G.D DO
    BEGIN
    WRITE(G@'D);
    FOR PP IN G@'T'INDEX(G@'D) DO
      WRITE(PP);
    END;
  END
```

The commands available to the system user have the following meanings:

TYPES	list the type definitions in the current form of the program.
REPS <type>	list the representations from the library which match the given type.
EVAL	evaluate the storage and execution time for the current form of the program.
PRINT	print the complete current form of the program.
QUIT	terminate the session.
SPLIT...	see subsection 5.4.2.

The user has printed the current form of the program, which consists of the initial abstract program augmented by the system-introduced GLOBAL type (see 5.3.4).

```
>REPS TABLE
CURRENTLY STOR=  10650B
CHOICE FOR TABLE
   ARRAY              STOR=   37362B
   SPARRAY            STOR=   11256B
   HASH               STOR=   11330B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
```

Here, a request has been made for the system to list the feasible representations for the type TABLE. The system responds with an indication of the minimum storage it computes to be necessary for the current form of the program (10650 bits), and then goes on to list the representations it has matched from the library. In this case three representations, with the identifiers ARRAY, SPARRAY and HASH, have been matched, and the system has evaluated the new minimal storage used if each were to be implemented. The user is invited to name one of the fully matched representations to implement in the program.

```
REP ARRAY;
   TYPE A=ARRAY B OF C; => TYPE A=ARRAY B OF C;
      GLOBAL B3:B;
   CONDITION CARD(B)<100000 AND MAXCOMP(A)>0;
   OPERATIONS
   S) AX"A:=A#ALL(CX"C) => FOR FB %CARD(B)% DO AX.INDEX(B3):=CX;
   R) AX"A.INDEX(BX"B)  => AX.INDEX(BX);
   V) AX"A'INDEX(BX"B)  => AX'INDEX(BX);
ENDREP
```

The ARRAY representation listed above, which simply allows a sparse array (as the program type TABLE is in this case) to be represented as a non-sparse array, is chosen by the user for implementation. (The outcome of choosing differently at this point is considered in 5.4.3).

```
>ARRAY

      TYPE GLOBAL=(BB01:DAY;P:PERSON;D:DAY;T:TABLE);
      TYPE PERSON=1..5000;
      TYPE DAY=1..366;
      TYPE TABLE=ARRAY DAY OF GROUP;
      TYPE GROUP=SET OF PERSON   (MAXCOMP=10 AVCOMP=1 );
```

The system carries out the implementation by transforming the internally stored program according to the representation description, and prints out the type specifications of the new form of the program.

These differ from the original in two respects in this case. Firstly, the global variable BB from the ARRAY representation has been included in the GLOBAL type as an additional component with selector BB01, derived from the variable name by adding a qualifying pair of digits to ensure uniqueness. Secondly, the MAXCOMP and AVCOMP qualifiers no longer appear on type TABLE, showing that the array is no longer considered as being sparse.

The user now requests that representations for the type GROUP be matched.

```
>REPS GROUP
CURRENTLY STOR=  37362B
CHOICE FOR GROUP
  POW1           STOR=1834801B
  POW2           STOR=  47643B
  INDIRECT       STOR=  40666B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>POW2

      TYPE GLOBAL=(SP02:INT;BB01:DAY;P:PERSON;D:DAY;T:TABLE);
      TYPE DAY=1..366;
      TYPE PERSON=1..5000;
      TYPE TABLE=ARRAY DAY OF GROUP;
      TYPE GROUP=SEQUENCE OF PERSON  (MAXCOMP=10 AVCOMP=1 );
```

The system matches three possible choices, from which the user selects the POW2 representation to be implemented, which represents the type as a sequence in place of a set.

```
>REPS GROUP
CURRENTLY STOR=  47643B
CHOICE FOR GROUP
  SEQ1           STOR=  49139B
  INDIRECT       STOR=  50946B
  SEQ2           STOR=  62681B
  SEQ3           STOR=  78419B
  SEQ5           STOR=  96011B
  SEQ6           STOR=  52433B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SEQ1
```

The user again asks to choose a representation for type GROUP, and now six possible representations are found to match the sequence. The user chooses the one with the least storage requirements, SEQ1. This representation has the following structure transformation:

```
REP SEQ1;
  TYPE A=SEQUENCE OF B; =>
    TYPE A=(FRST:E; AA:C);
    TYPE C=ARRAY D OF B;
    TYPE D=1..MAXCOMP(A);
    TYPE E=0..MAXCOMP(A);
    GLOBAL I:INT;
```

hence new types with appropriate specifications will be introduced with identifiers C03, D03, E03 and a new component I03 is added to the GLOBAL type for the additional global variable (03 being the pair of qualifying digits associated with this representation implementation).

The user prints out the current form of the program at this stage, which shows the new types as well as the changes that have been made to the data operations since the original abstract program was printed.

```
>PRINT
TYPE GLOBAL=(I03:INT;SP02:INT;BB01:DAY;P:PERSON;D:DAY;T:TABLE);
TYPE DAY=1..366;
TYPE PERSON=1..5000;
TYPE TABLE=ARRAY DAY OF GROUP;
TYPE GROUP=(FRST03:E03;AA03:C03);
TYPE C03=ARRAY D03 OF PERSON;
TYPE D03=1..10;
TYPE E03=0..10;
VAR G:GLOBAL;
  BEGIN
  FOR G.BB01 DO
    G.T.INDEX(G@'BB01).FRST03:=(0);
  READ(G.P);
  WHILE G@'P¬=0 DO
    BEGIN
    READ(G.D);
    G.SP02:=(G@'T'INDEX(G@'D)'FRST03);
    WHILE (¬(G@'SP02<1 OR G@'SP02>10)
              AND G@'T'INDEX(G@'D)'AA03'INDEX(G@'SP02)¬=G@'P) DO
      G.SP02:=((G@'SP02 - 1));
    IF (G@'SP02<1 OR G@'SP02>10) THEN
      BEGIN
      G.T.INDEX(G@'D).FRST03:=((G@'T'INDEX(G@'D)'FRST03 + 1));
      G.T.INDEX(G@'D).AA03.INDEX(G@'T'INDEX(G@'D)'FRST03):=(G@'P);
      END;
    READ(G.P);
    END;
  FOR G.D DO
    BEGIN
    WRITE(G@'D);
    FOR G.I03:1..G@'T'INDEX(G@'D)'FRST03 DO
      WRITE(G@'T'INDEX(G@'D)'AA03'INDEX(G@'I03));
    END;
  END
```

Two integer subrange types PERSON and E03 are next chosen in turn, and each is represented, using SUBREP, as the basic type int.

```
>REPS PERSON
CURRENTLY STOR=  49139B
CHOICE FOR PERSON
  SUBREP            STOR= 118698B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBREP

     TYPE GLOBAL=(I03:INT;SP02:INT;BB01:DAY;P:INT;D:DAY;T:TABLE);
     TYPE DAY=1..366;
     TYPE TABLE=ARRAY DAY OF GROUP;
     TYPE GROUP=(FRST03:E03;AA03:C03);
     TYPE E03=0..10;
     TYPE C03=ARRAY D03 OF INT;
     TYPE D03=1..10;
>REPS E03
CURRENTLY STOR= 118698B
CHOICE FOR E03
  SUBREP            STOR= 128946B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBREP

     TYPE GLOBAL=(I03:INT;SP02:INT;BB01:DAY;P:INT;D:DAY;T:TABLE);
     TYPE DAY=1..366;
     TYPE TABLE=ARRAY DAY OF GROUP;
     TYPE GROUP=(FRST03:INT;AA03:C03);
     TYPE C03=ARRAY D03 OF INT;
     TYPE D03=1..10;
```

Both of these representations, the first in particular, show how the minimum storage requirements can increase when a representation is applied which is 'wasteful' of space. In the case of type PERSON the subrange 1..5000 theoretically requires much less space than the 32 bits presumed for type int, however the savings in execution time of operations for accessing the latter type make the application of the representation worthwhile.

In order to progress further towards obtaining the desired target language structures, it is necessary to apply certain transformations to the program which are not strictly data representations, but transform a structure to an equivalent form that is more suited to subsequent transformations. In the experimental system, these kinds of transformations were written as if they were representations so that they could be stored in the representation library, selected and implemented by the same means as more conventional representations.

One of these equivalence transformations, given the name SYN2 in the library, has the following structure transformation:

```
REP SYN2;
  TYPE A=(BB:B;CC:C);  TYPE C=ARRAY D OF B;  TYPE D=M..N;
   => TYPE A=ARRAY E OF B;  TYPE E=M..N+1;  TYPE D=M..N;
```

It shows the equivalence between a Cartesian product, with one component being an array with elements of the same type as the other component of the product, and a single array with all elements of that type.

```
>REPS GROUP
CURRENTLY STOR= 128946B
CHOICE FOR GROUP
   SYNCP              STOR= 128946B
   SYN2               STOR= 128946B
   IN REP INDIRECT      FOLLOWING NOT IMPLEMENTED:-
      G.T.INDEX(G@*BP01).FRST03
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SYN2

      TYPE GLOBAL=(I03:INT;SP02:INT;BB01:DAY;P:INT;D:DAY;T:TABLE);
      TYPE DAY=1..366;
      TYPE TABLE=ARRAY DAY OF GROUP;
      TYPE GROUP=ARRAY E06 OF INT;
      TYPE E06=1..11;
      TYPE D03=1..10;
```

The SYN2 equivalence has been applied to type GROUP, so that it now takes the form of a single array, allowing a further equivalence to be applied to type TABLE. (The system indicates above a representation which failed to match, in order to aid analysis of its action).

```
>REPS TABLE
CURRENTLY STOR= 128946B
CHOICE FOR TABLE
   SYNAR              STOR= 128956B
   SYN5               STOR= 128946B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SYN5

      TYPE GLOBAL=(I03:INT;SP02:INT;BB01:DAY;P:INT;D:DAY;T:TABLE);
      TYPE DAY=1..366;
      TYPE TABLE=ARRAY F07 OF INT;
      TYPE F07=(BB07:DAY;DD07:E06);
      TYPE E06=1..11;
```

The application of SYN5 converts type TABLE to a single array of integers, but with a Cartesian product index type. This type, F07, is a Cartesian product of two subranges, and so can be converted to a single subrange by the SUBCP representation.

```
>REPS F07
CURRENTLY STOR= 128946B
CHOICE FOR F07
   IN REP SYNCP        FOLLOWING NOT IMPLEMENTED:-
      F07#(G@'BB01,11)
   SUBCP            STOR= 128946B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBCP

     TYPE GLOBAL=(I03:INT;SP02:INT;BB01:DAY;P:INT;D:DAY;T:TABLE);
     TYPE DAY=1..366;
     TYPE TABLE=ARRAY F07 OF INT;
     TYPE F07=0..4025;
     TYPE E06=1..11;
>REPS DAY
CURRENTLY STOR= 128946B
CHOICE FOR DAY
   SUBREP           STOR= 128992B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBREP

     TYPE GLOBAL=(I03:INT;SP02:INT;BB01:INT;P:INT;D:INT;T:TABLE);
     TYPE TABLE=ARRAY F07 OF INT;
     TYPE F07=0..4025;
```

Finally, type DAY is represented as an int to achieve a set of type specifications which conform to the requirements of the target language. The user requests an evaluation of the final form of the program:

```
 >EVAL
 CURRENTLY STOR= 128992B      TIME=  1.08'+05
```

which shows that it requires storage of 128992 bits (i.e. 4031 int cells), and that the predicted execution time is 1.08 x 10^5 µs (for the basic operation costs preset in the system for the target language).

The final internal form of the program can now be printed, showing the structures and transformed program in its concrete target language compatible form, and the representational choice session is terminated.

```
>PRINT
TYPE GLOBAL=(I03:INT;SP02:INT;BB01:INT;P:INT;D:INT;T:TABLE);
TYPE TABLE=ARRAY F07 OF INT;
TYPE F07=0..4025;
VAR G:GLOBAL;
  BEGIN
  FOR G.BB01:1..366 DO
    G.T.INDEX((G@'BB01+3659)):=(0);
  READ(G.P);
  WHILE G@'P¬=0 DO
    BEGIN
    READ(G.D);
    G.SP02:=(G@'T'INDEX((G@'D+3659)));
    WHILE (¬(G@'SP02<1 OR G@'SP02>10)
           AND G@'T'INDEX((((G@'SP02-1)*366)+(G@'D-1)))¬=G@'P) DO
      G.SP02:=((G@'SP02-1));
    IF (G@'SP02<1 OR G@'SP02>10) THEN
      BEGIN
      G.T.INDEX((G@'D+3659)):=((G@'T'INDEX((G@'D+3659))+1));
      G.T.INDEX((((G@'T'INDEX((G@'D+3659))-1)*366)+(G@'D-1))):=(G@'P);
      END;
    READ(G.P);
    END;
  FOR G.D:1..366 DO
    BEGIN
    WRITE(G@'D);
    FOR G.I03:1..G@'T'INDEX((G@'D+3659)) DO
      WRITE(G@'T'INDEX((((G@'I03-1)*366)+(G@'D-1))));
    END;
  END
```

By performing some trivial editing (which includes changing the single instance of type GLOBAL back into a set of variable declarations), the final concrete program can be transliterated into the syntax of a language such as AlgolW, and this program can be compiled and executed as normal.

```
BEGIN INTEGER SP02,P,D;
INTEGER ARRAY T(0::4025);
  BEGIN
  FOR BB01:=1 UNTIL 366 DO
    T((BB01+3659)):=(0);
  READ(P);
  WHILE P¬=0 DO
    BEGIN
    READ(D);
    SP02:=(T((D+3659)));
    WHILE (¬(SP02<1 OR SP02>10)
           AND T((((SP02-1)*366)+(D-1)))¬=P) DO
      SP02:=((SP02-1));
    IF (SP02<1 OR SP02>10) THEN
      BEGIN
      T((D+3659)):=((T((D+3659))+1));
      T((((T((D+3659))-1)*366)+(D-1))):=(P);
      END;
    READ(P);
    END;
  FOR D:=1 UNTIL 366 DO
    BEGIN
    WRITE(D);
    FOR I03:=1 UNTIL T((D+3659)) DO
      WRITE(T((((I03-1)*366)+(D-1))));
    END;
  END
END.
```

5.4.2 Equivalence transformations.

In the example session shown in the previous section two transformations were used which express equivalences among different data structures. These transformations were included in the representation library as pseudo-representations SYN2 and SYN5. It was found necessary to include a set of these transformations (SYN1 to SYN5 shown in Appendix III) to express such structure equivalences, and also two others (SYNCP and SYNAR) which express equivalences among operations for two structuring methods. By writing these equivalences in the form of representations and including them in the representation library, it was possible to use the same matching, selecting and implementing routines for them as for the more conventional representations.

One further kind of equivalence was not found to be expressible in the pseudo-representation form, but the necessity for its inclusion in the representation process required that a special command be included to allow the system user to invoke its application. The need for this equivalence arose from the desire to apply representations and equivalences to Cartesian product and discriminated union structures with more than two components. The form of representation used in the experimental system cannot express representations for these structures having arbitrary numbers of components, and the representations in the library include only two-component structures in their 'old' structure specifications. It is possible to extend the application of these representations to many-component Cartesian products and discriminated unions by repeatedly splitting two-component parts from the many-component structure. This can be achieved by using the equivalence:

```
    type A = (s1:A1;s2:A2;s3:A3; ... ;sn:An);  =>
                 type A = (s:B;s3:A3; ... ;sn:An);
                 type B = (s1:A1;s2:A2);
```

There are associated equivalences among the operations that can be applied to each structure. A special system command was introduced to perform this transformation:

```
SPLIT <type> <sel1> <sel2>
```

which splits from the named type a type with a two-component structure having the given selectors. Subsequently, the type with the two-component structure can be represented as normal, and if necessary further two-component parts can be split from the remaining type.

5.4.3 Options in guiding use.

The experimental system was implemented so that the user must decide what to do next at any given moment during the choice process. The two kinds of decision which affect the action of the system on the program are, firstly, the choice of which representation to implement when given a set of matched possibilities, and secondly the order in which representations are to be attempted for the various data types current at a given stage in the choice.

The first kind of decision could be made automatically if the evaluation of different representations was of use as a guide. The second kind of decision is one for which it appears difficult to give guidelines, whether in order to carry it out automatically or to assist the user to get best results from the system. The decisions made are important because they can affect the represenations applicable at different stages in the choice, and hence the resultant representations used in the final program.

In some instances, choosing and implementing a representation for type A, then one for type B, will result in exactly the same program as the case when B is implemented before A. However, in other cases, particularly when one of A or B is a structural component of the other, the implementation of one in a particular way first may rule out a subsequent representation for the other.

Example: In the example session of subsection 5.4.1, if type PERSON had been represented as an <u>int</u> using SUBREP before investigating the representation of type GROUP, it would then have been impossible to use the BITS representation for GROUP.

As an example of the choices open to the user, and the way in which the storage evaluation acts as a possible guide to selection, in Figure 5.4 we summarise some of the paths that can be taken to arrive at a concrete program for the Birthdays problem. Each path shows a sequence of representational choices for the various types in the program. At each branching node in the tree, the storage evaluation provided by the system is shown for each branch. At the leaves of the tree, denoting possible concrete programs, a full evaluation can be performed by the system for both storage space and execution time. Only a small number of the possible sequences of representational choice have been shown.

It can be seen that the sixth route minimises the storage cost from among the eight routes, and that the minimal execution times are given by the second and fourth routes, though these use appreciably more storage. (These evaluations depend on the target language costs that are preset in the system, and the relative merits of the different options may change if the basic costs are altered).

The example session of 5.4.1 followed the second route through the tree, selecting the 'array' representation for type TABLE, 'pow2' for type GROUP, and 'seq1' for type GROUP in turn (plus further selections not shown in the figure) to give a concrete program requiring 126k bits of storage and 0·11 seconds execution time.

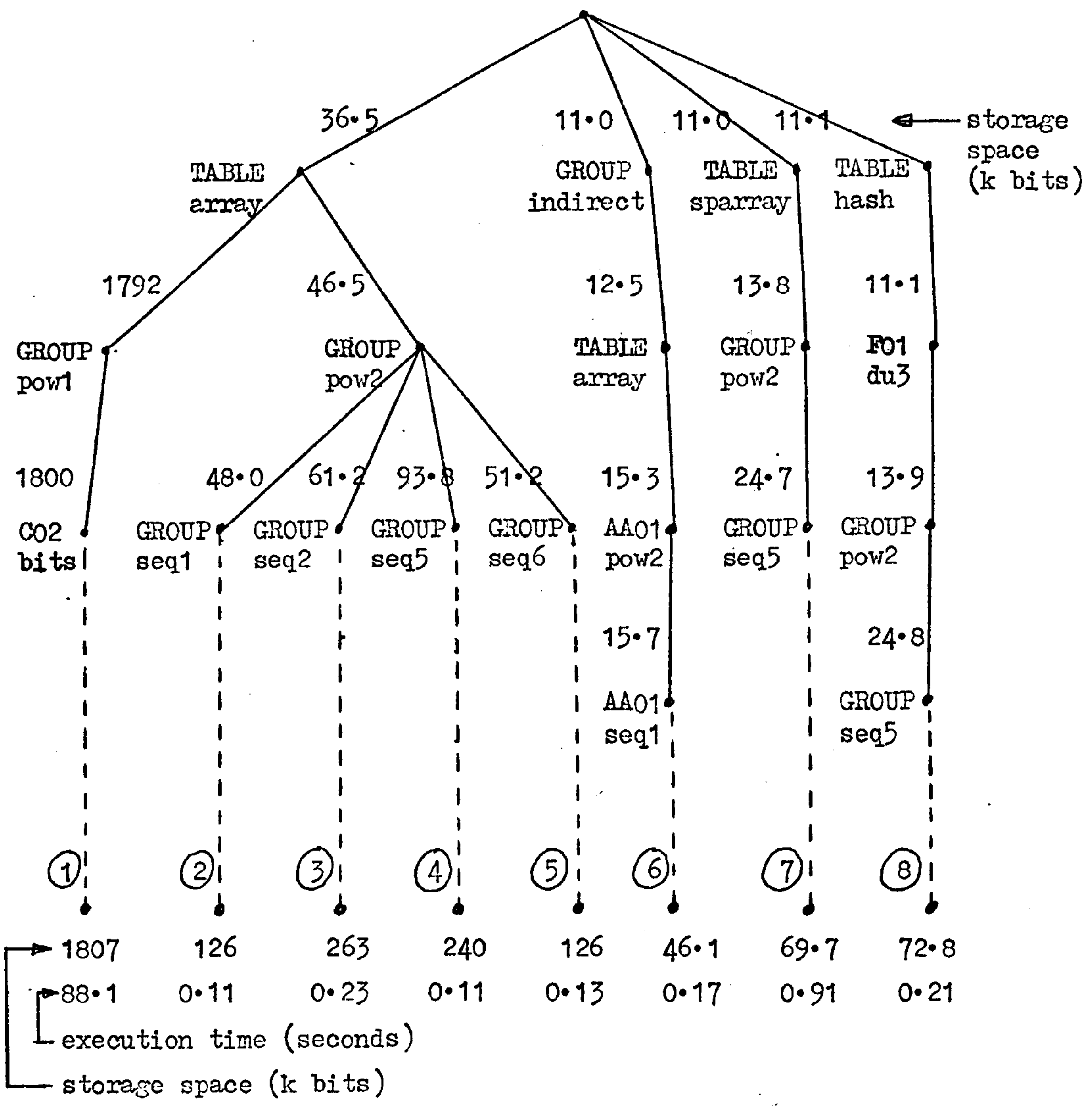


Figure 5.4 Paths to a concrete program, and evaluation, for Birthdays problem

It can be seen from the intermediate storage evaluation figures, provided by the system along the various choice routes, that they do provide a limited relative indication of the final storage costs. However, the earlier evaluations give little indication of the final absolute costs for any given route. Where such evaluation may be of help is in eliminating routes, such as route one in the figure, which will lead to unacceptably high storage costs, and therefore reduce the representational choice space that need be searched.

CHAPTER 6

ASSESSMENT OF TRANSFORMATIONAL APPROACH

6.0 Summary.

In this chapter we assess the transformational approach to data representation that has been described and demonstrated in the foregoing chapters. The original objective was to investigate techniques that would help the programmer in the selection and implementation of data structures during program development. The approach that was developed during the course of the investigations and which we have presented here will be critically evaluated to see how far it contributes towards the original objectives.

The assessment first considers the benefits of the approach, which were largely the motivation for the direction taken in the research. It then goes on to discuss the limitations to the approach that were discovered, and other ways in which the approach might be extended but which could not be further investigated within the extent of the present work.

Finally we conclude by summarising the main results that can be drawn from the investigations.

6.1 Benefits of the Approach.

The expression of data representations in terms of program transformations as demonstrated in the present work has several benefits which will be considered in turn.

6.1.1 Concise notation.

The notation developed and used for the investigations has allowed a range of data representations to be written in a uniform and concise fashion. Some of the representations which were written during the course of experimentation are shown in Appendix III. It was found that the method of presentation of the representations, separating the structure and operations, and fitting the latter into the categories described in section 4.3, provided a useful framework which enabled the form of a new representation to be developed in a thorough manner. At the same time, the notation is concise allowing the main features of a representation to be more quickly isolated, and it enables essential simple representations, which form the building bricks for more complicated representational transformation sequences, to be individually expressed.

6.1.2 Selection and implementation.

The selection of representations to be used in programs is aided by the approach. Firstly the range of choice from which representations may be chosen is made more explicit, and it is therefore less easy to overlook feasible representations. Secondly the selection of feasible representations is helped, particularly when machine assistance, of the form of the experimental system used here, is available. The representation description clearly distinguishes the nature of the data structure, and those of its associated operations, which it is designed to represent. Matching feasible representations therefore becomes straightforward.

Having the representation described in a machine-manipulable form also helps when implementing a chosen representation, since errors introduced by manual implementation can be avoided, and the implementation effort is considerably reduced.

6.1.3 Building of libraries.

The concise form of the notation, and the general applicability of the relatively small number of representations that were used during experimentation, make it appear feasible that useful libraries of reasonably limited numbers of data representations could be constructed. However, only more extensive experiments could determine whether the large range of representations currently in general use could be obtained from various combinations of entries in a relatively small library.

6.1.4 Re-use of software.

The programming involved in writing the implementing code for a representation can be used to its maximum by the approach, since a fixed set of implementations can be re-used many times in different programs, or even in the same program. If the implementation is thoroughly tested, or even proved to be correct, then when used with machine applied transformations, the re- use of such software could increase the reliability of those significant portions of programs which are concerned with supporting data representations.

6.2. Limitations and Extensions of the Approach

The transformational approach to data representation as presented here has raised certain further questions, and there are other limitations to the present approach that were intentionally imposed to restrict the investigations to an appropriate time and scope. In addition there are various extensions that would be desirable in order to make the use of the approach more feasible in a production environment, but which would require further investigation.

6.2.1 Selection and evaluation.

The original objective of the work was to provide the programmer with assistance in the choice of data representations, but not to take away his control over those representations which could be selected. This objective has been achieved to a large extent. However, as the example and discussion in section 5.4. has shown, the control that must be exercised by the programmer in directing the action of the experimental system is fairly extensive. This implies that a reasonably close knowledge of the working of the system is required to make effective use of its capabilities.

It is a matter for further consideration as to how much more assistance could be automatically provided in order that a less experienced user could benefit from the system. One direction in which more help might be given is in making a closer link between the evaluation provided by the system and the control over the investigation of sequences of transformation. A heuristic approach to limit the space of representations under consideration might be possible, but involves further investigation beyond the scope of the present work.

6.2.2. More than one representation per type.

One of the limitations of the approach adopted in these experiments is that each type in a program is restricted to a single representation. Whenever a representation is chosen for a specific type, all items and operations belonging to the type must be implemented in terms of the new structure and operations of the representation. If, however, there are, for example, two data items which conceptually have the same structure, but which are used in different ways in a program, they might be better represented in different way. To do this in the experimental system each item would have to be defined to be of a different type.

> Example: In the card game example, two kinds of pile of cards are defined:
>
> <u>type</u> PILE1 = <u>sequence</u> <u>of</u> PLAYING-CARD (MAXCOMP=52, AVCOMP=16)
> <u>type</u> PILE2 = <u>sequence</u> <u>of</u> PLAYING-CARD (MAXCOMP=52, AVCOMP=30)
>
> The two separate types, which however have the same conceptual structure, can be given separate usage information as shown, and allow variables of each type to be declared and used in separate ways. They can also possibly have separate representations chosen for them.

Among reasons for wanting to relax the rule of one representation per type, two can be illustrated, both of which have inherent complications.

Firstly, consider the situation described above, where data items of conceptually the same type might require the use of different representations, and where it may not be easy for the programmer to allocate each a different type beforehand. Attempting to overcome this by using a different representation for each data item rather than one representation per data type leads to complications when there are data operations that have operations of the same type but using different representations.

Example: Given the declarations

<u>type</u> PILE = <u>sequence</u> <u>of</u> PLAYING-CARD;
<u>var</u> face-up : PILE; <u>var</u> face-down : PILE;

and if different representations are chosen for the variables face-up and face-down because of their different usage in the program, then an operation such as

```
    face-up := face-down;
```

would involve a difficult representation-transfer implementation.

The second reason for relaxing the one representation per type rule is in the case of a single data item whose usage is very different in different parts of a program. In this case a different representation might be best for each program part. Complications arise here in deciding at what points a representation change becomes suitable, and, as in the previous case, in carrying out such a change in representation.

6.2.3 Representation of more flexible data.

The form of program used during the experiments has been of the most simple kind, limiting the data to be globally and statically declared, and not considering the division of the program into sub-components such as procedures. These limitations were imposed in order to keep the scope of the investigations within reasonable bounds. It is likely that any practical application would require more flexible data to be considered.

Such flexibility could include block structured programs and associated storage allocation, the inclusion of procedures with parameters of types whose representation were under consideration, and the extension of representational choice to input/output data. All these possibilities involve non-trivial extensions beyond the present investigations, and could prove difficult or even impossible without changes to the approach.

Though the approach presented here is applicable to recursively defined data structures, the experimental system was not extended to support such structures, and so practical experiments on their representation were not carried out. Handling such structures, where their manipulation often involves recursive procedures, raises the same kind of problems as inclusion of procedures for program subdivision.

6.2.4 Further investigations.

Besides the problem areas and limitations that have already been pointed out as requiring more study, many more topics associated with data representation could be singled out for further investigation. We will give some examples of the peripheral areas that have arisen during these studies.

The structuring methods used in the experimental system were selected from those used by Hoare in his Notes on Data Structuring [25] Further structuring methods could be included, in particular the extension to include the n-ary relation as a structure would significantly increase the range of expressible structures, especially to include those often found in data-base applications. The representation of data in data-bases is currently a topic of interest, and much work has been published on it. The automatic design of the data organisation for such applications has been considered by McCuskey [41] The use of relations as structuring methods in high-level programming languages has been explored by Earley [16] , Feldman [17] and others.

The question of orderings on the component values of data structures, such as the components of a sequence, and also orderings on the values of a data type, such as type PLAYING-CARD denoting the cards in a pack of playing cards, have been largely ignored in the current investigations. Considerations of such orderings would probably affect the representation chosen for the data. For example an ordered sequence might be represented as an ordered binary tree, so that insertion of a component value into the correct position could be performed more easily.

Fitting the representations chosen in a program to the particular data types and structuring methods available in a given target language is another area of possible further investigation. The choice of which representations to make available in a library is likely to be affected by whether or not, say, the language includes the Cartesian product (record) as a structuring method. If it does not include the structure, representations must be available to implement a Cartesian product in terms, such as an array, that the language does support.

6.3 Conclusions

The investigations reported here have aimed to show the requirements for and feasibility of program transformations as an approach to assisting data structure representation. In previous chapters the concepts that were developed in order to express such representations in a machine-manipulable form have been described, and the experiences gained from an experimental system embodying the concepts have been presented.

The main results obtained from the investigations can be summarised as follows

- We have exposed the concepts needed to express data representations as program transformations, and embodied these concepts in a notation which allows such transformations to be expressed in a concise but readable form.
- We have used the notation to write a library of data representations which enable many commonly encountered traditional representations to be enumerated and made available for use in programs.
- We have shown how many representations can be expressed as various combinations of the basic transformations included in the library, allowing the size of the library to be kept within reasonable limits while still providing a large range of possible representations.
- We have designed and implemented an experimental system to aid in the selection and implementation of representations, which has enabled the identification of those features which would be desirable in a practical representational choice system.
- We have demonstrated how the experimental system can be used to select and implement representations for some example programs.

The field of data structures and their representation is a very large one that permeates all forms of computing. As such, the investigations reported here have made only a small, but still significant, contribution towards making the task of data representation somewhat easier. Inevitably perhaps, many aspects of providing assistance with the choice of representation have been shown to require further investigation. However, only by attempting the kind of study reported here can progress be made towards a better understanding of the concepts involved.

APPENDIX I

SUMMARY OF THE STRUCTURING METHODS AND TYPES USED IN THE EXPERIMENTS

The following structuring methods were adopted for the specification of data types in the experimental system. For each structuring method are given the syntax used in declarations of types which have that structure, the calculation of the cardinality of such a type, the data operations that were defined on operations belonging to that type, and the type property functions (used in writing representations) which apply to that type.

In addition to the operations specific to each structuring method that are given below, the contents operation @, the assignment update operation := and the equality operator = are applicable to data items of any type.

In addition to the specific property functions that are given below, the following property functions apply to any type T:

CARD(T)	cardinality of T.
STOR(T)	storage required by a value of type T ($=\lceil \log_2 \mathrm{CARD}(T) \rceil$).
MAXNUM(T)	maximum number of items of type T used in program.
AVNUM(T)	average number of items of type T used in program.

a) Integer Subrange

Syntax: type T = m..n

Cardinality: CARD(T) = n-m+1

Operations:

T#(iv) construct a value of type T from int value iv.

T#INT(tv) construct an int value from a T value tv.

(The above two operations which perform type-conversion between subrange values and basic type int values can be inferred from context in writing programs and representations, and need not be explicitly shown in system input and output).

for tr do S repeat statement S for tr taking each value of type T in turn.

b) Cartesian Product

Syntax: *type* T = (s1:T1; ... ;sn:Tn)

Cardinality: CARD(T) = CARD(T1) x ... x CARD(Tn)

Operations:

T#(t1v, ... ,tnv)	construct a value of type T from component values.
tr.si	reference-selection of the ith component.
tv'si	value-selection of the ith component.
for tr *do* S	repeat statement S for tr taking each value of type T in turn.

c) Discriminated Union

Syntax: *type* T = (k1:T1 ... kn:Tn)

Cardinality: CARD(T) = CARD(T1) + ... + CARD(Tn)

Operations:

T#ki(tiv)	construct a value with the ki alternate value tiv.
tv?ki	test whether tv is currently the ki alternate.
tr.ki	reference-select component of a ki alternate.
tv'ki	value-select component of a ki alternate.

d) Array

Syntax: *type* T = *array* T1 *of* T2

Cardinality: $\mathrm{CARD(T)} = \mathrm{CARD(T2)}^{\mathrm{CARD(T1)}}$ (see also subsection 5.3.4)

Operations:

T#ALL(t2v)	construct array with all elements having value t2v.
tr.INDEX(t1v)	reference-select array element indexed by t1v.
tv'INDEX(t1v)	value-select array element indexed by t1v.

Properties: MAXCOMP(T), AVCOMP(T) maximum and average numbers of elements in arrays of type T which differ from a common default (sparse arrays).

e) Set

Syntax: *type* T = *set of* T1

Cardinality: $\mathrm{CARD(T)} = 2^{\mathrm{CARD(T1)}}$ (see also subsection 5.3.4)

Operations:

T#EMPTY	construct an empty set value.
tr:INSERT(t1v)	insert element with value t1v into set tr.
tr:REMOVE(t1v)	remove " " " " " " "
tv?HAS(t1v)	test whether set tv contains value t1v.
tv?EMPTY	test whether set tv is empty.
tv?NUMEL	return number of elements currently in set tv.
<u>for</u> t1v <u>in</u> tv <u>do</u> S	repeat S with t1v taking value of each element of tv in turn.

Properties: MAXCOMP(T), AVCOMP(T) maximum and average numbers of elements in sets of type T.

f) <u>Sequence</u>

Syntax: <u>type</u> T = <u>sequence</u> <u>of</u> T1

Cardinality: ∞ (but see subsection 5.3.4)

Operations:

T#EMPTY	construct empty sequence value.
tr:APPEND_FIRST(t1v)	append t1v to beginning of sequence tr.
tr:APPEND_LAST(t1v)	" " " end " " "
tr:REMOVE_FIRST	remove first element from sequence tr.
tr:REMOVE_LAST	" last " " " "
tr.FIRST	reference-select first element of tr.
tr.LAST	" last " " "
tv'FIRST	value-select first element of tr.
tv'LAST	" last " " "
tv?EMPTY	test whether sequence tv is empty.
tv?NUMEL	return number of elements in sequence tv.
<u>for</u> t1v <u>in</u> tv <u>do</u> S	repeat S for t1v taking values of each element of sequence from first to last in turn.

The following sequence operations allow a 'pointer' of type <u>int</u> to be used to denote a position in the sequence for selecting, inserting and removing elements.

tv?FIRST	returns <u>int</u> pointer to first element.
tv?LAST	" " " " last "
tv?NEXT(pv)	returns pointer to next element after that indicated by pointer pv.
tv?PREV(pv)	returns pointer to previous element before that indicated by pointer pv.

tv?NOCURR(pv)	tests whether pv does not indicate a currently valid element of the sequence.
tr.CURR(pv)	reference-selection of element pointed to by pv.
tv'CURR(pv)	value-selection of element pointed to by pv.
tr:INSERT_NEXT(pv,t1v)	insert value t1v after element pointed to by pv.
tr:INSERT_PREV(pv,t1v)	insert value t1v before element pointed to by pv.
tr:REMOVE_CURR(pv)	remove element pointed to by pv.

Properties: MAXCOMP(T), AVCOMP(T) maximum and average numbers of elements in sequences of type T.

In the experimental system three basic types were included with the following characteristics.

1) Basic type: <u>int</u>
Storage: 32 bits
Values: $-2^{31} .. 2^{31}-1$
Operations: +,-,*,<u>div</u>,<u>rem</u>,=,¬=,>,<,>=,<= as in conventional languages

ir:SETBIT(jv,lv)	the integer denoted by ir has the bit in position jv set to 1 or 0 depending on whether the logical value lv is true or false.
iv'BIT(jv)	returns true or false depending on whether bit jv is set to 1 or 0.

2) Basic type: <u>logical</u>
Cardinality: 2
Values: <u>true</u>,<u>false</u>
Operations: AND,OR,NOT as in conventional languages.

3) Basic type: <u>null</u>
Cardinality: 1
Value: <u>nil</u>
Operations: none.

APPENDIX II

EXAMPLE PROGRAMS

The following listings show the abstract programs written for the two example problems used in the main text. These are in the form that they are input to the experimental choice system, including the programmer-provided usage information about the number of components in structures and the frequency of execution of loops etc.

Abstract program for Birthdays Problem

```
TYPE TABLE=ARRAY DAY OF GROUP (AVCOMP=50,MAXCOMP=100);
TYPE DAY=1..366;
TYPE GROUP=SET OF PERSON (AVCOMP=1,MAXCOMP=10);
TYPE PERSON=1..5000;
VAR T:TABLE; VAR D:DAY; VAR P:PERSON;
  BEGIN
  T:=TABLE#ALL(GROUP#EMPTY);
  READ(P);
  WHILE P¬=0 %50.% DO
    BEGIN READ(D);
    T.INDEX(D):INSERT(P);
    READ(P);
    END;
  FOR D %366.% DO
    BEGIN WRITE(D);
    FOR PP IN T'INDEX(D) %1.5% DO WRITE(PP);
    END;
  END
.
```

Abstract program for Card Game Problem

```
TYPE PILE1=SEQUENCE OF PLAYING_CARD (MAXCOMP=52,AVCOMP=16);
TYPE PILE2=SEQUENCE OF PLAYING_CARD (MAXCOMP=52,AVCOMP=30);
TYPE PLAYERS=ARRAY PLAYER OF HAND;
TYPE PLAYER=1..2;
TYPE HAND=SET OF PLAYING_CARD (MAXCOMP=52,AVCOMP=3);
TYPE PLAYING_CARD=(SUIT:SUIT:RANK:RANK);
TYPE RANK=1..13;  TYPE SUIT=1..4;
VAR PLAYERS:PLAYERS; VAR PLAYER:PLAYER;
VAR FACE_UP:PILE1; VAR FACE_DOWN:PILE2;
VAR P:PLAYER; VAR PLAYED:LOGICAL;
VAR S:SUIT; VAR R:RANK; VAR C:PLAYING_CARD; VAR I:INT;
   BEGIN
   FACE_UP:=PILE1#EMPTY; FACE_DOWN:=PILE2#EMPTY;
   PLAYERS:=PLAYERS#ALL(HAND#EMPTY);
   FOR I:0..51 %52.% DO
      FACE_DOWN:APPEND_FIRST(PLAYING_CARD#(I DIV 13+1,I REM 13+1));
   FOR P %2.% DO
      FOR I:1..7 %7.% DO
         BEGIN
         PLAYERS.INDEX(P):INSERT(FACE_DOWN'FIRST);
         FACE_DOWN:REMOVE_FIRST;
         END;
   FACE_UP:APPEND_FIRST(FACE_DOWN'FIRST);
   FACE_DOWN:REMOVE_FIRST;
   PLAYER:=1;
   WHILE ¬(PLAYERS'INDEX(1)?EMPTY OR PLAYERS'INDEX(2)?EMPTY) %30.% DO
      BEGIN PLAYED:=FALSE; S:=1;
      WRITE(FACE_UP'FIRST'RANK); WRITE(FACE_UP'FIRST'SUIT);
      WHILE ¬PLAYED AND S<=4 %3.76% DO
         BEGIN
         C:=PLAYING_CARD#(S,FACE_UP'FIRST'RANK);
         IF PLAYERS'INDEX(PLAYER)?HAS(C) %0.06% THEN
            BEGIN
            FACE_UP:APPEND_FIRST(C);
            PLAYERS.INDEX(PLAYER):REMOVE(C);
            PLAYED:=TRUE;
            END
         ELSE S:=S+1;
         END;
      R:=1;
      WHILE ¬PLAYED AND R<=13 %6.% DO
         BEGIN
         C:=PLAYING_CARD#(FACE_UP'FIRST'SUIT,R);
         IF PLAYERS'INDEX(PLAYER)?HAS(C) %0.06% THEN
            BEGIN
            FACE_UP:APPEND_FIRST(C);
            PLAYERS.INDEX(PLAYER):REMOVE(C);
            PLAYED:=TRUE;
            END
         ELSE R:=R+1;
         END;
      IF ¬PLAYED %0.57% THEN
         BEGIN
         WRITE(PLAYER);
         WRITE(FACE_DOWN'FIRST'RANK); WRITE(FACE_DOWN'FIRST'SUIT);
         PLAYERS.INDEX(PLAYER):INSERT(FACE_DOWN'FIRST);
         FACE_DOWN:REMOVE_FIRST;
         END;
      PLAYER:=PLAYER REM 2 +1;
      END;
   WRITE(PLAYERS'INDEX(1)?EMPTY);
   END
```

APPENDIX III

EXAMPLE REPRESENTATION LIBRARY

The following listings show the representations that were included in the library used during the experimentation. The representations are divided into two groups, the first group being those that provide the conventional representational transformations, and the second group being pseudo-representations that define equivalences between different structures and their operations and whose use was illustrated in the example of section 5.4.

The choice of operations that are implemented for each representation and the evaluation information that is provided in each representation are both at the discretion of the writer of the representation. No detailed analysis has been carried out on how appropriately these factors have been covered in the example library, except that the choice of operations has proved adequate for the experiments carried out using the library.

For ease of syntactic analysis in the experimental system, two conventions were used in writing the operation transformations. Firstly, each operation transformation is preceded by the annotation S), R) or V) depending on whether the operation being transformed takes the form of a statement, a data reference or a data value respectively. Secondly for each formal parameter on the left hand side of an operation transformation, the type of that parameter is given by appending it with a double quote after the parameter identifier. Thus AX"A denotes a parameter of type A with identifier AX.

```
REP SUBREP;
   TYPE A=M..N; => TYPE INT;
   CONDITION CARD(A)<CARD(INT);
   OPERATIONS
   V)   A#INT(AX"A)  => AX;
   V)   A#(I"INT)  => I;
   S)   FOR AX"A DO S"STATEMENT => FOR AX:M..N %CARD(A)% DO S;
ENDREP
```

SUBREP : subrange represented as a basic <u>int</u> type

```
REP SUBCP;
   TYPE A=(BB:B;CC:C); TYPE B=K..L; TYPE C=M..N; =>
     TYPE A=0..CARD(A)-1: TYPE B=K..L; TYPE C=M..N;
   CONDITION CARD(A)<CARD(INT);
   OPERATIONS
   V) AX"A'BB => AX REM %L-K+1% +K;
   V) AX"A'CC => AX DIV %L-K+1% +M;
   V) A#(BX"B,CX"C) => (CX-M)*%L-K+1% +(BX-K);
   S) AX"A.BB:=BX"B => AX:=(AX DIV %L-K+1%)*%L-K+1% +(BX-K);
   S) AX"A.CC:=CX"C => AX:=(CX-M)*%L-K+1% +(AX REM %L-K+1%);
ENDREP
```

SUBCP : Cartesian product packed into a single subrange

```
REP DU1;
   TYPE A=(K1:NULL|K2:B); TYPE B=M..N; =>
     TYPE A=M..N+1; TYPE B=M..N;
   OPERATIONS
   V) A#K1(NIL) => %N+1%;
   V) A#K2(BX"B) => BX;
   V) AX"A'K1 => NIL;
   V) AX"A'K2 => AX;
   V) AX"A?K1 => AX=%N+1%;
   V) AX"A?K2 => AX<%N+1%;
ENDREP
```

DU1 : discriminated union with null alternative packed into a single subrange

```
REP DU3;
   TYPE A=(K1:NULL|K2:B); =>
     TYPE A=(ALT:C;V:B); TYPE C=0..1;
   OPERATIONS
   S) AX"A:=A#K1(NIL) => AX.ALT:=0;
   S) AX"A:=A#K2(BX"B) => BEGIN AX.ALT:=0; AX.V:=BX END;
   R) AX"A.K2 => AX.V;
   V) AX"A'K1 => NIL;
   V) AX"A'K2 => AX'V;
   V) AX"A?K1 => AX'ALT=0;
   V) AX"A?K2 => AX'ALT=1;
   V) A#K2(BX"B) => A#(1,BX);
ENDREP
```

DU3 : discriminated union with null alternative represented with a separate tag field

```
REP SUBDU1;
  TYPE A=(BB:B|CC:C); TYPE B=K..L; TYPE C=M..N; =>
    TYPE A=K..N; TYPE B=K..L; TYPE C=M..N;
  CONDITION L<M;
  OPERATIONS
  V) AX"A'BB => AX;
  V) AX"A'CC => AX;
  V) AX"A?BB => AX<=L;
  V) AX"A?CC => AX>=M;
  V) A#BB(BX"B) => BX;
  V) A#CC(CX"C) => CX;
ENDREP
```

SUBDU1 : disjoint discriminated union packed into a single subrange

```
REP SUBDU2;
  TYPE A=(BB:B|CC:C); TYPE B=K..L; TYPE C=M..N; =>
    TYPE A=0..CARD(B)+CARD(C)-1; TYPE B=K..L; TYPE C=M..N;
  CONDITION CARD(A)<CARD(INT);
  OPERATIONS
  V) AX"A'BB => AX+K;
  V) AX"A'CC => AX-%CARD(B)-M%;
  V) AX"A?BB => AX<CARD(B);
  V) AX"A?CC => AX>=CARD(B);
  V) A#BB(BX"B) => BX-K;
  V) A#CC(CX"C) => CX+%CARD(B)-M%;
ENDREP
```

SUBDU2 : non-disjoint discriminated union packed into a subrange

```
REP ARRAY;
  TYPE A=ARRAY B OF C; => TYPE A=ARRAY B OF C;
    GLOBAL BB:B;
  CONDITION CARD(B)<100000 AND MAXCOMP(A)>0;
  OPERATIONS
  S) AX"A:=A#ALL(CX"C) => FOR BB %CARD(B)% DO AX.INDEX(BB):=CX;
  R) AX"A.INDEX(BX"B) => AX.INDEX(BX);
  V) AX"A'INDEX(BX"B) => AX'INDEX(BX);
ENDREP
```

ARRAY : sparse array expanded into a non-sparse array

```
REP BITS;
  TYPE A=ARRAY B OF LOGICAL; TYPE B=M..N; =>
    TYPE A=ARRAY C OF INT; TYPE C=0..(N-M) DIV 32; TYPE B=M..N;
    GLOBAL CC:C;
  OPERATIONS
  V) AX"A'INDEX(BX"B) => AX'INDEX((BX-M) DIV 32)'BIT((BX-M) REM 32);
  S) AX"A.INDEX(BX"B):=L"LOGICAL =>
       AX.INDEX((BX-M) DIV 32):SETBIT((BX-M) REM 32,L);
  S) AX"A:=A#ALL(TRUE)  => FOR CC %(N-M) DIV 32% DO AX.INDEX(CC):=0-1;
  S) AX"A:=A#ALL(FALSE) => FOR CC %(N-M) DIV 32% DO AX.INDEX(CC):=0;
  V) A#ALL(TRUE)  => A#ALL(0-1);
  V) A#ALL(FALSE) => A#ALL(0);
ENDREP
```

BITS : array of bits represented as array of <u>int</u>

```
REP HASH;
  TYPE A=ARRAY B OF C; TYPE B=M..N; =>
    TYPE A=(DEF:C;REST:D);
    TYPE D=ARRAY E OF F;
    TYPE E=0..MAXCOMP(A)-1;
    TYPE F=(NONE:NULL|SOME:G);
    TYPE G=(IND:B:VAL:C);
    TYPE B=M..N;
    GLOBAL K:E;
  CONDITION MAXCOMP(A)>0;
  OPERATIONS
  S) AX"A:=A#ALL(CX"C) =>
       BEGIN AX.DEF:=CX;
       FOR K %MAXCOMP(A)% DO AX.REST.INDEX(K):=F#NONE(NIL);
       END;
  V) A#ALL(CX"C) => A#(CX,D#ALL(F#NONE(NIL)));
  V) AX"A'INDEX(BX"B) =>
       BEGIN K:=BX REM MAXCOMP(A);
       WHILE ¬AX'REST'INDEX(K)?NONE
         AND ¬AX'REST'INDEX(K)'SOME'IND=BX %0.8% DO
           K:=(K+1) REM MAXCOMP(A);
       RESULT IF AX'REST'INDEX(K)?NONE
                   %1-AVCOMP(A)/CARD(A)% THEN AX'DEF
                ELSE AX'REST'INDEX(K)'SOME'VAL  END;
  S) AX"A.INDEX(BX"B):=CX"C =>
       BEGIN K:=BX REM MAXCOMP(A);
       WHILE ¬AX'REST'INDEX(K)?NONE
         AND ¬AX'REST'INDEX(K)'SOME'IND=BX %0.8% DO
           K:=(K+1) REM MAXCOMP(A);
       AX.REST.INDEX(K):=F#SOME(G#(BX,CX))
       END;
  S) AX"A.INDEX(BX"B):ANYOP =>
       BEGIN K:=BX REM MAXCOMP(A):
       WHILE ¬AX'REST'INDEX(K)?NONE
         AND ¬AX'REST'INDEX(K)'SOME'IND=BX %0.8% DO
           K:=(K+1) REM MAXCOMP(A);
       IF AX'REST'INDEX(K)?NONE %AVCOMP(A)/2.% THEN
         BEGIN AX.REST.INDEX(K):=F#SOME(G#(BX,AX'DEF));
         AX.REST.INDEX(K).SOME.VAL:ANYOP; END
       ELSE AX.REST.INDEX(K).SOME.VAL:ANYOP;
       END;
ENDREP
```

HASH : sparse array hashed into non-sparse array

```
REP SPARRAY;
  TYPE A=ARRAY B OF C; =>
    TYPE A=(DEF:C;REST:D);
    TYPE D=SEQUENCE OF E (MAXCOMP=MAXCOMP(A),AVCOMP=AVCOMP(A));
    TYPE E=(IND:B;VAL:C);
    GLOBAL SP:INT;
  CONDITION MAXCOMP(A)>0;
  OPERATIONS
  S) AX"A:=A#ALL(CX"C) => BEGIN AX.DEF:=CX; AX.REST:=D#EMPTY END;
  V) A#ALL(CX"C) => A#(CX,D#EMPTY);
  S) AX"A.INDEX(BX"B):=CX"C =>
       IF ¬(CX=AX'DEF) %AVCOMP(A)/CARD(B)% THEN
         BEGIN SP:=AX'REST?FIRST;
         WHILE ¬AX'REST?NOCURR(SP) AND ¬(AX'REST'CURR(SP)'IND=BX)
           %AVCOMP(A)*0.8% DO
           SP:=AX'REST?NEXT(SP);
         IF AX'REST?NOCURR(SP) %1-AVCOMP(A)/CARD(B)% THEN
           AX.REST:APPEND_FIRST(E#(BX,CX))
         ELSE AX.REST.CURR(SP).VAL:=CX;
         END;
  S) AX"A.INDEX(BX"B):ANYOP =>
       BEGIN SP:=AX'REST?FIRST;
       WHILE ¬AX'REST?NOCURR(SP) AND ¬(AX'REST'CURR(SP)'IND=BX)
         %AVCOMP(A)*0.8% DO
         SP:=AX'REST?NEXT(SP);
       IF AX'REST?NOCURR(SP) %1-AVCOMP(A)/CARD(B)% THEN
         BEGIN AX.REST:APPEND_FIRST(E#(BX,AX'DEF));
         AX.REST.FIRST.VAL:ANYOP; END
       ELSE AX.REST.CURR(SP).VAL:ANYOP;
       END;
  V) AX"A'INDEX(BX"B) =>
       BEGIN SP:=AX'REST?FIRST;
       WHILE ¬AX'REST?NOCURR(SP) AND ¬(AX'REST'CURR(SP)'IND=BX)
         %AVCOMP(A)*0.8% DO
         SP:=AX'REST?NEXT(SP);
       RESULT IF AX'REST?NOCURR(SP) %1-AVCOMP(A)/CARD(B)%THEN AX'DEF
              ELSE AX'REST'CURR(SP)'VAL    END;
ENDREP
```

SPARRAY : sparse array as default plus sequence of non-defaults

```
REP POW1;
  TYPE A=SET OF B; =>
    TYPE A=(NUMBEL:D;EL:C);
    TYPE C=ARRAY B OF LOGICAL;
    TYPE D=0..CARD(B); GLOBAL BB:B;
  CONDITION CARD(B)<100000;
  OPERATIONS
  S) AX"A:INSERT(BX"B) =>
       BEGIN AX.EL.INDEX(BX):=TRUE;
       AX.NUMBEL:=AX.NUMBEL+1;
       END;
  S) AX"A:REMOVE(BX"B) =>
       BEGIN AX.EL.INDEX(BX):=FALSE;
       AX.NUMBEL:=AX.NUMBEL-1;
       END;
  V) AX"A?EMPTY => AX'NUMBEL=0;
  S) AX"A:=A#EMPTY =>
       BEGIN
       AX.NUMBEL:=0;
       AX.EL:=C#ALL(FALSE);
       END;
  V) A#EMPTY => A#(0,C#ALL(FALSE));
  S) FOR BX IN AX"A DO S"STATEMENT =>
       FOR BB %CARD(B)% DO
         IF AX'EL'INDEX(BB) %AVCOMP(A)/CARD(B)% THEN S WITH BX=BB;
  V) AX"A?HAS(BX"B) => AX'EL'INDEX(BX);
  V) AX"A?NUMEL => AX'NUMBEL;
ENDREP
```

POW1 : set represented as array of bits

```
REP POW2;
 TYPE A=SET OF B; =>
    TYPE A=SEQUENCE OF B (MAXCOMP=MAXCOMP(A),AVCOMP=AVCOMP(A));
    GLOBAL SP:INT;
  OPERATIONS
  S) AX"A:INSERT(BX"B) =>
       BEGIN SP:=AX?FIRST;
       WHILE ¬AX?NOCURR(SP) AND AX'CURR(SP)¬=BX %AVCOMP(A)*0.8% DO
         SP:=AX?NEXT(SP);
       IF AX?NOCURR(SP) %0.8% THEN AX:APPEND_FIRST(BX);
       END;
  S) AX"A:REMOVE(BX"B) =>
       BEGIN SP:=AX?FIRST;
       WHILE ¬AX?NOCURR(SP) AND AX'CURR(SP)¬=BX %AVCOMP(A)*0.8% DO
         SP:=AX?NEXT(SP);
       IF ¬AX?NOCURR(SP) %0.8% THEN AX:REMOVE_CURR(SP);
       END;
  V) AX"A?HAS(BX"B) =>
       BEGIN SP:=AX?FIRST;
       WHILE ¬AX?NOCURR(SP) AND AX'CURR(SP)¬=BX %AVCOMP(A)*0.8% DO
         SP:=AX?NEXT(SP);
       RESULT ¬AX?NOCURR(SP) END;
  V) AX"A?EMPTY => AX?EMPTY;
  V) AX"A?NUMEL => AX?NUMEL;
  V) A#EMPTY => A#EMPTY;
  S) FOR BX IN AX"A DO S"STATEMENT =>
      FOR BX IN AX %AVCOMP(A)% DO S;
ENDREP
```

POW2 : set represented as sequence

```
REP SEQ1;
  TYPE A=SEQUENCE OF B; =>
    TYPE A=(FRST:E; AA:C);
    TYPE C=ARRAY D OF B;
    TYPE D=1..MAXCOMP(A);
    TYPE E=0..MAXCOMP(A);
    GLOBAL I:INT;
  OPERATIONS
  R) AX"A.FIRST  => AX.AA.INDEX(AX'FRST);
  V) AX"A'FIRST  => AX'AA'INDEX(AX'FRST);
  R) AX"A.LAST   => AX.AA.INDEX(1);
  V) AX"A'LAST   => AX'AA'INDEX(1);
  V) AX"A?EMPTY  => AX'FRST=0;
  V) AX"A?NUMEL  => AX'FRST;
  S) AX"A:APPEND_FIRST(BX"B) =>
       BEGIN AX.FRST:=AX'FRST+1;
       AX.AA.INDEX(AX'FRST):=BX; END;
  S) AX"A:REMOVE_FIRST => AX.FRST:=AX'FRST-1;
  S) AX"A:REMOVE_CURR(SP"INT) =>
       BEGIN AX.FRST:=AX'FRST-1;
       FOR I:SP..AX'FRST %AVCOMP(A)/2.% DO
         AX.AA.INDEX(I):=AX'AA'INDEX(I+1);
       END;
  S) AX"A:APPEND_LAST(BX"B) =>
       BEGIN AX.FRST:=AX'FRST+1;
       FOR I:AX'FRST..2 %AVCOMP(A)% DO
         AX.AA.INDEX(I):=AX'AA'INDEX(I-1);
       AX.AA.INDEX(1):=BX;
       END;
  S) AX"A:INSERT_NEXT(SP"INT,BX"B) =>
       BEGIN AX.FRST:=AX'FRST+1;
       FOR I:AX'FRST..SP+1 %AVCOMP(A)/2.% DO
         AX.AA.INDEX(I):=AX'AA'INDEX(I-1);
       AX.AA.INDEX(SP):=BX;
       END;
  S) AX"A:INSERT_PREV(SP"INT,BX"B) =>
       BEGIN AX.FRST:=AX'FRST+1;
       FOR I:AX'FRST..SP+2 %AVCOMP(A)/2.% DO
         AX.AA.INDEX(I):=AX'AA'INDEX(I-1);
       AX.AA.INDEX(SP+1):=BX;
       END;
  V) AX"A?FIRST  => AX'FRST;
  V) AX"A?LAST   => 1;
  V) AX"A?NEXT(SP"INT) => SP-1;
  V) AX"A?PREV(SP"INT) => SP+1;
  V) AX"A?NOCURR(SP"INT) => SP<1 OR SP>MAXCOMP(A);
  R) AX"A.CURR(SP"INT) => AX.AA.INDEX(SP);
  V) AX"A'CURR(SP"INT) => AX'AA'INDEX(SP);
  S) AX"A:=A#EMPTY => AX.FRST:=0;
  S) FOR BX IN AX"A DO S"STATEMENT =>
       FOR I:1..AX'FRST %AVCOMP(A)% DO S WITH BX=AX'AA'INDEX(I);
ENDREP
```

SEQ1 : sequence as contiguous array with first element pointer

```
 REP SEQ2;
   TYPE A=SEQUENCE OF B; =>
     TYPE A=(FRST:E;FREE:E;NUEL:E;AA:C);
     TYPE C=ARRAY D OF F;
     TYPE F=(VAL:B;NEX:E);
     TYPE D=1..MAXCOMP(A); TYPE E=0..MAXCOMP(A);
     GLOBAL I:INT;
  OPERATIONS
  S) AX"A:=A#EMPTY =>
        BEGIN AX.FRST:=0; AX.FREE:=1; AX.NUEL:=0;
        FOR I:1..MAXCOMP(A)-1 %MAXCOMP(A)-1% DO
          AX.AA.INDEX(I).NEX:=I+1;
        AX.AA.INDEX(MAXCOMP(A)).NEX:=0;
        END;
  R) AX"A.FIRST => AX.AA.INDEX(AX'FRST).VAL;
  V) AX"A'FIRST => AX'AA'INDEX(AX'FRST)'VAL;
  V) AX"A?EMPTY => AX'FRST=0;
  V) AX"A?NUMEL => AX'NUEL;
  S) AX"A:APPEND_FIRST(BX"B) =>
        BEGIN I:=AX'FREE; AX.FREE:=AX'AA'INDEX(AX'FREE)'NEX;
        AX.AA.INDEX(I):=F#(BX,AX'FRST);
        AX.FRST:=I;. AX.NUEL:=AX'NUEL+1;
        END;
  S) AX"A:REMOVE_FIRST =>
        BEGIN I:=AX'AA'INDEX(AX'FRST)'NEX;
        AX.AA.INDEX(AX'FRST).NEX:=AX'FREE;
        AX.FREE:=AX'FRST;
        AX.FRST:=I; AX.NUEL:=AX'NUEL+1;
        END;
  V) AX"A?FIRST => AX'FRST;
  V) AX"A?NEXT(SP"INT) => AX'AA'INDEX(SP)'NEX;
  V) AX"A?NOCURR(SP"INT) => SP=0;
  R) AX"A.CURR(SP"INT) => AX.AA.INDEX(SP).VAL;
  V) AX"A'CURR(SP"INT) => AX'AA'INDEX(SP)'VAL;
  S) AX"A:INSERT_NEXT(SP"INT,BX"B) =>
        BEGIN I:=AX'FREE; AX.FREE:=AX'AA'INDEX(AX'FREE)'NEX;
        AX.AA.INDEX(I):=F#(BX,AX'AA'INDEX(SP)'NEX);
        AX.AA.INDEX(SP).NEX:=I; AX.NUEL:=AX'NUEL+1;
        END;
  S) FOR BB IN AX"A DO S"STATEMENT =>
        BEGIN I:=AX'FRST;
        WHILE I¬=0 %AVCOMP(A)% DO
          BEGIN S WITH BB=AX'AA'INDEX(I)'VAL;
          I:=AX'AA'INDEX(I)'NEX;
          END;
        END;
ENDREP
```

SEQ2 : sequence as singly linked list

```
REP SEQ3;
  TYPE A=SEQUENCE OF B; =>
    TYPE A=(FRST:E;LST:E;FREE:E;NUEL:E;AA:C);
    TYPE C=ARRAY D OF F;
    TYPE F=(VAL:B;PRE:E;NEX:E);
    TYPE D=1..MAXCOMP(A); TYPE E=0..MAXCOMP(A);
    GLOBAL I:INT;
  OPERATIONS
  S) AX"A:=A#EMPTY =>
       BEGIN AX.FRST:=0; AX.LST:=0; AX.FREE:=1; AX.NUEL:=0;
       FOR I:1..MAXCOMP(A)-1 %MAXCOMP(A)-1% DO
          AX.AA.INDEX(I).NEX:=I+1;
       AX.AA.INDEX(MAXCOMP(A)).NEX:=0;
       END;
  R) AX"A.FIRST => AX.AA.INDEX(AX'FRST).VAL;
  V) AX"A'FIRST => AX'AA'INDEX(AX'FRST)'VAL;
  R) AX"A.LAST => AX.AA.INDEX(AX'LST).VAL;
  V) AX"A'LAST => AX'AA'INDEX(AX'LST)'VAL;
  V) AX"A?EMPTY => AX'FRST=0;
  V) AX"A?NUMEL => AX'NUEL;
  S) AX"A:APPEND_FIRST(BX"B) =>
       BEGIN I:=ĀX'FREE; AX.FREE:=AX'AA'INDEX(AX'FREE)'NEX;
       AX.AA.INDEX(I):=F#(BX,0,AX'FRST);
       AX.FRST:=I; AX.NUEL:=AX'NUEL+1;
       END;
  S) AX"A:APPEND_LAST(BX"B) =>
       BEGIN I:=ĀX'FREE; AX.FREE:=AX'AA'INDEX(AX'FREE)'NEX;
       AX.AA.INDEX(I):=F#(BX,AX'LST,0);
       AX.LST:=I; AX.NUEL:=AX'NUEL+1;
       END;
  S) AX"A:REMOVE_FIRST =>
       BEGIN I:=ĀX'AA'INDEX(AX'FRST)'NEX;
       AX.AA.INDEX(AX'FRST).NEX:=AX'FREE;
       AX.FREE:=AX'FRST;
       AX.FRST:=I; AX.NUEL:=AX'NUEL-1;
       END;
  S) AX"A:REMOVE_LAST =>
       BEGIN I:=ĀX'AA'INDEX(AX'LST)'PRE;
       AX.AA.INDEX(AX'LST).NEX:=AX'FREE;
       AX.FREE:=AX'LST; AX.NUEL:=AX'NUEL-1; AX.LST:=I;
       END;
  V) AX"A?FIRST => AX'FRST;
  V) AX"A?LAST  => AX'LST;
  V) AX"A?NEXT(SP"INT) => AX'AA'INDEX(SP)'NEX;
  V) AX"A?PREV(SP"INT) => AX'AA'INDEX(SP)'PRE;
  V) AX"A?NOCURR(SP"INT) => SP=0;
  R) AX"A.CURR(SP"INT) => AX.AA.INDEX(SP).VAL;
  V) AX"A'CURR(SP"INT) => AX'AA'INDEX(SP)'VAL;
  S) AX"A:INSERT_NEXT(SP"INT,BX"B) =>
       BEGIN I:=ĀX'FREE; AX.FREE:=AX'AA'INDEX(AX'FREE)'NEX;
       AX.AA.INDEX(I):=F#(BX,SP,AX'AA'INDEX(SP)'NEX);
       AX.AA.INDEX(SP).NEX:=I; AX.NUEL:=AX'NUEL+1;
       END;
  S) AX"A:INSERT_NEXT(SP"INT,BX"B) =>
       BEGIN I:=ĀX'FREE; AX.FREE:=AX'AA'INDEX(AX'FREE)'NEX;
       AX.AA.INDEX(I):=F#(BX,AX'AA'INDEX(SP)'PRF,SP);
       AX.AA.INDEX(SP).PRE:=I; AX.NUEL:=AX'NUEL+1;
       END;
  S) AX"A:REMOVE_CURR(SP"INT) =>
       BEGIN
       AX.AA.INDEX(AX'AA'INDEX(SP)'PRE).NEX:=AX'AA'INDEX(SP)'NEX;
       AX.AA.INDEX(AX'AA'INDEX(SP)'NEX).PRE:=AX'AA'INDEX(SP)'PRE;
       AX.AA.INDEX(SP).NEX:=AX'FREE; AX.FREE:=SP;
       AX.NUEL:=AX'NUEL-1;
       END;
  S) FOR BB IN AX"A DO S"STATEMENT =>
       BEGIN I:=AX'FRST;
       WHILE I¬=0 %AVCOMP(A)% DO
          BEGIN S WITH BB=AX'AA'INDEX(I)'VAL;
          I:=AX'AA'INDEX(I)'NEX;
          END;
       END;
ENDREP
```

SEQ3 : sequence as doubly linked list

```
REP SEQ5;
  TYPE A=SEQUENCE OF B; =>
    TYPE A=0..MAXCOMP(A)*MAXNUM(A);
    TYPE G=(LASTU:A;STORE:H);
    TYPE H=ARRAY M OF K;
    TYPE K=(VAL:B;NEX:A);
    TYPE M=1..MAXCOMP(A)*MAXNUM(A);
    GLOBAL ST:G; GLOBAL I:A; GLOBAL J:INT;
  OPERATIONS
  S) INIT => ST.LASTU:=0;
  V) A#EMPTY => 0;
  V) AX"A?EMPTY => AX=0;
  V) AX"A?NUMEL =>
       BEGIN J:=0; I:=AX;
       WHILE I¬=0 %AVCOMP(A)% DO
         BEGIN J:=J+1; I:=ST'STORE'INDEX(I)'NEX END;
       RESULT J END;
  S) AX"A:APPEND_FIRST(BX"B) =>
       BEGIN ST.LASTU:=ST'LASTU+1;
       I:=AX; AX:=ST'LASTU;
       ST.STORE.INDEX(AX):=K#(BX,I);
       END;
  S) AX"A:REMOVE_FIRST => AX:=ST'STORE'INDEX(AX)'NEX;
  R) AX"A.FIRST => ST.STORE.INDEX(AX).VAL;
  V) AX"A'FIRST => ST'STORE'INDEX(AX)'VAL;
  V) AX"A?FIRST => AX;
  V) AX"A?NEXT(SP"INT) => ST'STORE'INDEX(SP)'NEX;
  V) AX"A?NOCURR(SP"INT) => SP=0;
  R) AX"A.CURR(SP"INT) => ST.STORE.INDEX(SP).VAL;
  V) AX"A'CURR(SP"INT) => ST'STORE'INDEX(SP)'VAL;
  S) AX"A:INSERT_NEXT(SP"INT,BX"B) =>
       BEGIN ST.LASTU:=ST'LASTU+1;
       ST.STORE.INDEX(ST'LASTU):=K#(BX,ST'STORE'INDEX(SP)'NEX);
       ST.STORE.INDEX(SP).NEX:=ST'LASTU;
       END;
  S) FOR BX IN AX"A DO S"STATEMENT =>
       BEGIN I:=AX;
       WHILE I¬=0 %AVCOMP(A)% DO
         BEGIN S WITH BX=ST'STORE'INDEX(I)'VAL;
         I:=ST'STORE'INDEX(I)'NEX;
         END;
       END;
ENDREP
```

SEQ5 : several sequences as singly linked lists in common storage

```
REP SEQ6;
  TYPE A=SEQUENCE OF B; =>
    TYPE A=ARRAY C OF D;
    TYPE C=0..MAXCOMP(A);
    TYPE D=(LST:NULL|VAL:B);
    GLOBAL I:INT;
  OPERATIONS
  R)  AX"A.FIRST  =>  AX.INDEX(0).VAL;
  V)  AX"A'FIRST  =>  AX'INDEX(0)'VAL;
  R)  AX"A.LAST  =>
        BEGIN I:=0;
        WHILE AX'INDEX(I)?VAL  %AVCOMP(A)%  DO  I:=I+1;
        RESULT AX.INDEX(I-1).VAL END;
  V)  AX"A'LAST  =>
        BEGIN I:=0;
        WHILE AX'INDEX(I)?VAL  %AVCOMP(A)%  DO  I:=I+1;
        RESULT AX'INDEX(I-1)'VAL END;
  V)  AX"A?EMPTY  =>  AX'INDEX(0)?LST;
  V)  AX"A?NUMEL  =>
        BEGIN I:=0;
        WHILE AX'INDEX(I)?VAL  %AVCOMP(A)%  DO  I:=I+1;
        RESULT  I  END;
  S)  AX"A:=A#EMPTY  =>  AX.INDEX(0):=D#LST(NIL);
  V)  A#EMPTY        =>  A#ALL(D#LST(NIL));
  S)  AX"A:APPEND_FIRST(BX"B)  =>
        BEGIN  I:=0;
        WHILE AX'INDEX(I)?VAL  %AVCOMP(A)%  DO  I:=I+1;
        FOR  I:I+1..1  %AVCOMP(A)%  DO  AX.INDEX(I):=AX'INDEX(I-1);
        AX.INDEX(0):=D#VAL(BX);
        END;
  S)  AX"A:APPEND_LAST(BX"B)  =>
        BEGIN  I:=0;
        WHILE AX'INDEX(I)?VAL  %AVCOMP(A)%  DO  I:=I+1;
        AX.INDEX(I):=D#VAL(BX);
        AX.INDEX(I+1):=D#LST(NIL);
        END;
  S)  AX"A:REMOVE_FIRST  =>
        BEGIN  I:=0;
        WHILE AX'INDEX(I)?VAL  %AVCOMP(A)%  DO
          BEGIN AX.INDEX(I):=AX'INDEX(I+1);  I:=I+1  END;
        END;
  S)  AX"A:REMOVE_LAST  =>
        BEGIN  I:=0;
        WHILE AX'INDEX(I)?VAL  %AVCOMP(A)%  DO  I:=I+1;
        AX.INDEX(I-1):=D#LST(NIL);
        END;
  V)  AX"A?FIRST  =>  0;
  V)  AX"A?LAST  =>
        BEGIN  I:=0;
        WHILE AX'INDEX(I)?VAL  %AVCOMP(A)%  DO  I:=I+1;
        RESULT  I-1  END;
  V)  AX"A?NEXT(SP"INT)  =>  SP+1;
  V)  AX"A?NOCURR(SP"INT)  =>  AX'INDEX(SP)?LST;
  R)  AX"A.CURR(SP"INT)  =>  AX.INDEX(SP).VAL;
  V)  AX"A'CURR(SP"INT)  =>  AX'INDEX(SP)'VAL;
  S)  AX"A:INSERT_NEXT(SP"INT,BX"B)  =>
        BEGIN  I:=SP;
        WHILE AX'INDEX(I)?VAL  %AVCOMP(A)%  DO  I:=I+1;
        FOR  I:I+1..SP+2  %AVCOMP(A)/2.%  DO  AX.INDEX(I):=AX'INDEX(I-1);
        AX.INDEX(SP+1):=D#VAL(BX);
        END;
  S)  AX"A:REMOVE_CURR(SP"INT)  =>
        BEGIN  I:=SP;
        WHILE AX'INDEX(I)?VAL  %AVCOMP(A)%  DO
          BEGIN  AX.INDEX(I):=AX'INDEX(I+1);  I:=I+1  END;
        END;
  S)  FOR BX  IN  AX"A  DO  S"STATEMENT  =>
        BEGIN  I:=0;
        WHILE AX'INDEX(I)?VAL  %AVCOMP(A)%  DO
          BEGIN  S  WITH  BX=AX'INDEX(I)'VAL;
          I:=I+1;
          END;
        END;
ENDREP
```

SEQ6 : sequence as contiguous array with last element marker

```
REP INDIRECT:
  TYPE A=ANYSTRUC; =>
    TYPE A=1..MAXNUM(A);
    TYPE U=(LASTU:B; S:STORE);
    TYPE B=0..MAXNUM(A);
    TYPE STORE=ARRAY A OF AA;
    TYPE AA=ANYSTRUC (MAXCOMP=MAXCOMP(A), AVCOMP=AVCOMP(A));
    GLOBAL UU:U;
  CONDITION MAXNUM(A)>1 AND STOR(A)>STOR(INT);
  OPERATIONS
  S) INIT => UU.LASTU:=0;
  V) AX"A'ANYOP => UU'S'INDEX(AX)'ANYOP;
  V) AX"A?ANYOP => UU'S'INDEX(AX)?ANYOP;
  V) A#ANYOP =>
       BEGIN UU.LASTU:=UU'LASTU+1;
       UU.S.INDEX(UU'LASTU):=AA#ANYOP;
       RESULT UU'LASTU END;
  S) A1"A:=A2"A => A1:=A2;
  V) A1"A=A2"A => (A1=A2) OR (UU'S'INDEX(A1)=UU'S'INDEX(A2));
  S) AX"A:ANYOP =>
       BEGIN UU.LASTU:=UU'LASTU+1;
       UU.S.INDEX(UU'LASTU):=UU'S'INDEX(AX);
       UU.S.INDEX(UU'LASTU):ANYOP;
       AX:=UU'LASTU;
       END;
  S) FOR BX IN AX"A DO S"STATEMENT =>
       FOR BX IN UU'S'INDEX(AX) %AVCOMP(A)% DO S;
ENDREP
```

INDIRECT : all instances of a type in common storage with pointers

Equivalence transformations

```
REP SYNCP;
  TYPE A=(S1:B;S2:C);   =>  TYPE A=(S1:B;S2:C);
  OPERATIONS
  S) AX"A:=AY"A  => BEGIN AX.S1:=AY'S1; AX.S2:=AY'S2 END;
  V) AX"A=AY"A   => (AX'S1=AY'S1) AND (AX'S2=AY'S2);
  S) AX"A:=A#(BX"B,CX"C) => BEGIN AX.S1:=BX; AX.S2:=CX END;
  V) AX"A=A#(BX"B,CX"C)  => (AX'S1=BX) AND (AX'S2=CX);
  R) AX"A.S1 => AX.S1;
  R) AX"A.S2 => AX.S2;
  V) AX"A'S1 => AX'S1;
  V) AX"A'S2 => AX'S2;
ENDREP
```

```
REP SYNAR;
  TYPE A=ARRAY B OF C; => TYPE A=ARRAY B OF C;
    GLOBAL BB:B; GLOBAL EQ:LOGICAL;
  CONDITION MAXCOMP(A)=0;
  OPERATIONS
  S) AX"A:=AY"A => FOR BB %CARD(B)% DO AX.INDEX(BB):=AY'INDEX(BB);
  V) AX"A=AY"A   =>
       BEGIN EQ:=TRUE;
       FOR BB %CARD(B)% DO
         EQ:=EQ AND (AX'INDEX(BB)=AY'INDEX(BB));
       RESULT EQ END;
  S) AX"A:=A#ALL(CX"C) => FOR BB %CARD(B)% DO AX.INDEX(BB):=CX;
  R) AX"A.INDEX(BX"B) => AX.INDEX(BX);
  V) AX"A'INDEX(BX"B) => AX'INDEX(BX);
ENDREP
```

```
REP SYN1;
  TYPE A=(B1:B;B2:B); => TYPE A=ARRAY C OF B; TYPE C=0..1;
  OPERATIONS
  R) AX"A.B1 => AX.INDEX(0);
  R) AX"A.B2 => AX.INDEX(1);
  V) AX"A'B1 => AX'INDEX(0);
  V) AX"A'B2 => AX'INDEX(1);
ENDREP


REP SYN2;
  TYPE A=(BB:B;CC:C); TYPE C=ARRAY D OF B; TYPE D=M..N;
   => TYPE A=ARRAY E OF B; TYPE E=M..N+1; TYPE D=M..N;
  OPERATIONS
  R) AX"A.BB => AX.INDEX(%N+1%);
  R) AX"A.CC.INDEX(DD"D) => AX.INDEX(DD);
  V) AX"A'BB => AX'INDEX(%N+1%);
  V) AX"A'CC'INDEX(DD"D) => AX'INDEX(DD);
ENDREP


REP SYN3;
  TYPE A=(BB:B;CC:C); TYPE B=ARRAY D OF C; TYPE D=M..N;
   => TYPE A=ARRAY E OF C; TYPE E=M..N+1; TYPE D=M..N;
  OPERATIONS
  R) AX"A.CC => AX.INDEX(%N+1%);
  R) AX"A.BB.INDEX(DD"D) => AX.INDEX(DD);
  V) AX"A'CC => AX'INDEX(%N+1%);
  V) AX"A'BB'INDEX(DD"D) => AX'INDEX(DD);
ENDREP


REP SYN4;
  TYPE A=(BB:B;CC:C); TYPE B=ARRAY D OF E; TYPE C=ARRAY F OF E;
   => TYPE A=ARRAY G OF E; TYPE G=(DD:D|FF:F);
  OPERATIONS
  R) AX"A.BB.INDEX(DX"D) => AX.INDEX(G#DD(DX));
  R) AX"A.CC.INDEX(FX"F) => AX.INDEX(G#FF(FX));
  V) AX"A'BB'INDEX(DX"D) => AX'INDEX(G#DD(DX));
  V) AX"A'CC'INDEX(FX"F) => AX'INDEX(G#FF(FX));
ENDREP


REP SYN5;
  TYPE A=ARRAY B OF C; TYPE C=ARRAY D OF E;
   => TYPE A=ARRAY F OF E; TYPE F=(BB:B;DD:D);
  OPERATIONS
  R) AX"A.INDEX(BX"B).INDEX(DX"D) => AX.INDEX(F#(BX,DX));
  V) AX"A'INDEX(BX"B)'INDEX(DX"D) => AX'INDEX(F#(BX,DX));
  V) A#ALL(C#ALL(EX"E)) => A#ALL(EX);
ENDREP
```

APPENDIX IV

EXAMPLE REPRESENTATIONAL CHOICE SESSION

As another example of the use of the experimental system, besides that given in section 5.4, the following listing shows a complete sequence of representational choice made for the Card Game Problem. It is presumed that the target language includes single-dimensioned arrays as its only structuring method. The representations that are matched and implemented during the course of this session are all detailed in Appendix III.

```
** REPRESENTATIONAL CHOICE SYSTEM **
COMMANDS:
   TYPES, REPS <TYPE>, EVAL, PRINT, QUIT,
   SPLIT <TYPE> <SEL> <SEL>
>TYPES

     TYPE GLOBAL=(I:INT;C:PLAYING_CARD;R:RANK;S:SUIT;
       PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;FACE_UP:PILE1;
       PLAYER:PLAYER;PLAYERS:PLAYERS);
     TYPE PILE1=SEQUENCE OF PLAYING_CARD  (AVCOMP=16 MAXCOMP=52);
     TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
     TYPE PLAYERS=ARRAY PLAYER OF HAND;
     TYPE PLAYER=1..2;
     TYPE HAND=SET OF PLAYING_CARD  (AVCOMP=3 MAXCOMP=52);
     TYPE PLAYING_CARD=(SUIT:SUIT;RANK:RANK);
     TYPE RANK=1..13;
     TYPE SUIT=1..4;
>REPS PLAYING_CARD
CURRENTLY STOR=     813B
CHOICE FOR PLAYING_CARD
  IN REP SYNCP         FOLLOWING NOT IMPLEMENTED:-
     PLAYING_CARD#(((G@'I REM 13) + 1),((G@'I DIV 13) + 1))
  SUBCP           STOR=     813B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBCP

     TYPE GLOBAL=(I:INT;C:PLAYING_CARD;R:RANK;S:SUIT;
       PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;FACE_UP:PILE1;
       PLAYER:PLAYER;PLAYERS:PLAYERS);
     TYPE PLAYING_CARD=0..51;
     TYPE SUIT=1..4;
     TYPE RANK=1..13;
     TYPE PLAYER=1..2;
     TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
     TYPE PILE1=SEQUENCE OF PLAYING_CARD  (AVCOMP=16 MAXCOMP=52);
     TYPE PLAYERS=ARRAY PLAYER OF HAND;
     TYPE HAND=SET OF PLAYING_CARD  (AVCOMP=3 MAXCOMP=52);
>REPS HAND
CURRENTLY STOR=     813B
CHOICE FOR HAND
  POW1            STOR=     793B
  POW2            STOR=    1327B
  INDIRECT        STOR=     817B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>POW1

     TYPE GLOBAL=(BB02:PLAYING_CARD;I:INT;C:PLAYING_CARD;R:RANK;
       S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
       FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
     TYPE PLAYING_CARD=0..51;
     TYPE RANK=1..13;
     TYPE SUIT=1..4;
     TYPE PLAYER=1..2;
     TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
     TYPE PILE1=SEQUENCE OF PLAYING_CARD  (AVCOMP=16 MAXCOMP=52);
     TYPE PLAYERS=ARRAY PLAYER OF HAND;
     TYPE HAND=(NUMBEL02:D02;EL02:C02);
     TYPE C02=ARRAY PLAYING_CARD OF LOGICAL;
     TYPE D02=0..52;
>REPS C02
CURRENTLY STOR=     793B
CHOICE FOR C02
  IN REP SYNAR         FOLLOWING NOT IMPLEMENTED:-
     C02#ALL( FALSE)
  IN REP INDIRECT      FOLLOWING NOT IMPLEMENTED:-
     G.PLAYERS.INDEX(G@'P).EL02.INDEX(G@'FACE_DOWN'FIRST)
  BITS            STOR=     818B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>BITS
```

```
      TYPE GLOBAL=(CC03:C03;BB02:PLAYING_CARD;I:INT;C:PLAYING_CARD;
        R:RANK;S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
        FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE PLAYING_CARD=0..51;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PLAYER=1..2;
      TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
      TYPE PILE1=SEQUENCE OF PLAYING_CARD  (AVCOMP=16 MAXCOMP=52);
      TYPE PLAYERS=ARRAY PLAYER OF HAND;
      TYPE HAND=(NUMBEL02:D02;EL02:C02);
      TYPE D02=0..52;
      TYPE C02=ARRAY C03 OF INT;
      TYPE C03=0..1;
  >REPS D02
  CURRENTLY STOR=     818B
  CHOICE FOR D02
     SUBREP        STOR=     870B
  TO IMPLEMENT REP, GIVE CHOSEN REP NAME
  >SUBREP

      TYPE GLOBAL=(CC03:C03;BB02:PLAYING_CARD;I:INT;C:PLAYING_CARD;
        R:RANK;S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
        FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE C03=0..1;
      TYPE PLAYING_CARD=0..51;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PLAYER=1..2;
      TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
      TYPE PILE1=SEQUENCE OF PLAYING_CARD  (AVCOMP=16 MAXCOMP=52);
      TYPE PLAYERS=ARRAY PLAYER OF HAND;
      TYPE HAND=(NUMBEL02:INT;EL02:C02);
      TYPE C02=ARRAY C03 OF INT;
  >REPS PLAYERS
  CURRENTLY STOR=     870B
  CHOICE FOR PLAYERS
     SYNAR             STOR=     872B
  TO IMPLEMENT REP, GIVE CHOSEN REP NAME
  >SYNAR

      TYPE GLOBAL=(EQ05:LOGICAL;BB05:PLAYER;CC03:C03;
        BB02:PLAYING_CARD;I:INT;C:PLAYING_CARD;R:RANK;S:SUIT;
        PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;FACE_UP:PILE1;
        PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE C03=0..1;
      TYPE PLAYING_CARD=0..51;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PLAYER=1..2;
      TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
      TYPE PILE1=SEQUENCE OF PLAYING_CARD  (AVCOMP=16 MAXCOMP=52);
      TYPE PLAYERS=ARRAY PLAYER OF HAND;
      TYPE HAND=(NUMBEL02:INT;EL02:C02);
      TYPE C02=ARRAY C03 OF INT;
  >REPS HAND
  CURRENTLY STOR=     872B
  CHOICE FOR HAND
     SYNC2          STOR=      872B
     IN REP SYN2          FOLLOWING NOT IMPLEMENTED:-
        HAND#(0,C02#ALL(0))
     IN REP INDIRECT      FOLLOWING NOT IMPLEMENTED:-
        G.PLAYERS.INDEX(G?'P).EL02
  TO IMPLEMENT REP, GIVE CHOSEN REP NAME
  >SYNC2
```

```
      TYPE GLOBAL=(EQ05:LOGICAL;BB05:PLAYER;CC03:C03;
        BB02:PLAYING_CARD;I:INT;C:PLAYING_CARD;R:RANK;S:SUIT;
        PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;FACE_UP:PILE1;
        PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE PLAYER=1..2;
      TYPE C03=0..1;
      TYPE PLAYING_CARD=0..51;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
      TYPE PILE1=SEQUENCE OF PLAYING_CARD  (AVCOMP=16 MAXCOMP=52);
      TYPE PLAYERS=ARRAY PLAYER OF HAND;
      TYPE HAND=(NUMBEL02:INT;EL02:C02);
      TYPE C02=ARRAY C03 OF INT;
>REPS C02
CURRENTLY STOR=    872B
CHOICE FOR C02
   IN REP BAS1         FOLLOWING NOT IMPLEMENTED:-
      C02#ALL(0)
   SYNAR            STOR=    874B
   IN REP INDIRECT     FOLLOWING NOT IMPLEMENTED:-
      G.PLAYERS.INDEX(G@'P).EL02.INDEX((G@'FACE_DOWN'FIRST DIV 32))
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SYNAR

      TYPE GLOBAL=(EQ07:LOGICAL;BB07:C03;EQ05:LOGICAL;BB05:PLAYER;
        CC03:C03;BB02:PLAYING_CARD;I:INT;C:PLAYING_CARD;R:RANK;
        S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
        FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE PLAYER=1..2;
      TYPE C03=0..1;
      TYPE PLAYING_CARD=0..51;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
      TYPE PILE1=SEQUENCE OF PLAYING_CARD  (AVCOMP=16 MAXCOMP=52);
      TYPE PLAYERS=ARRAY PLAYER OF HAND;
      TYPE HAND=(NUMBEL02:INT;EL02:C02);
      TYPE C02=ARRAY C03 OF INT;
>REPS HAND
CURRENTLY STOR=    874B
CHOICE FOR HAND
   SYNCP            STOR=    874B
   SYN2             STOR=    874B
   IN REP INDIRECT     FOLLOWING NOT IMPLEMENTED:-
      G.PLAYERS.INDEX(G@'BB05).NUMBEL02
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SYN2

      TYPE GLOBAL=(EQ07:LOGICAL;BB07:C03;EQ05:LOGICAL;BB05:PLAYER;
        CC03:C03;BB02:PLAYING_CARD;I:INT;C:PLAYING_CARD;R:RANK;
        S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
        FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE C03=0..1;
      TYPE PLAYER=1..2;
      TYPE PLAYING_CARD=0..51;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
      TYPE PILE1=SEQUENCE OF PLAYING_CARD  (AVCOMP=16 MAXCOMP=52);
      TYPE PLAYERS=ARRAY PLAYER OF HAND;
      TYPE HAND=ARRAY E08 OF INT;
      TYPE E08=0..2;
>REPS PLAYERS
CURRENTLY STOR=    874B
CHOICE FOR PLAYERS
   SYNAR            STOR=    876B
   SYN5             STOR=    874B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SYN5
```

```
      TYPE GLOBAL=(EQ07:LOGICAL;BB07:C03;EQ05:LOGICAL;BB05:PLAYER;
        CC03:C03;BB02:PLAYING_CARD;I:INT;C:PLAYING_CARD;R:RANK;
        S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
        FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE C03=0..1;
      TYPE PLAYER=1..2;
      TYPE PLAYING_CARD=0..51;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
      TYPE PILE1=SEQUENCE OF PLAYING_CARD  (AVCOMP=16 MAXCOMP=52);
      TYPE PLAYERS=ARRAY F09 OF INT;
      TYPE F09=(BB09:PLAYER;DD09:E08);
      TYPE E08=0..2;
>REPS F09
CURRENTLY STOR=     874B
CHOICE FOR F09
   IN REP SYNCP       FOLLOWING NOT IMPLEMENTED:-
     F09#(G@'BB05,2)
  SUBCP            STOR=     874B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBCP

      TYPE GLOBAL=(EQ07:LOGICAL;BB07:C03;EQ05:LOGICAL;BB05:PLAYER;
        CC03:C03;BB02:PLAYING_CARD;I:INT;C:PLAYING_CARD;R:RANK;
        S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
        FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE C03=0..1;
      TYPE PLAYER=1..2;
      TYPE PLAYING_CARD=0..51;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
      TYPE PILE1=SEQUENCE OF PLAYING_CARD  (AVCOMP=16 MAXCOMP=52);
      TYPE PLAYERS=ARRAY F09 OF INT;
      TYPE F09=0..5;
      TYPE E08=0..2;
>REPS PILE1
CURRENTLY STOR=     874B
CHOICE FOR PILE1
  SEQ1             STOR=     912B
  SEQ2             STOR=    1236B
  SEQ3             STOR=    1553B
  SEQ5             STOR=    1236B
  SEQ6             STOR=     912B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SEQ1

      TYPE GLOBAL=(I11:INT;EQ07:LOGICAL;BB07:C07;EQ05:LOGICAL;
        BB05:PLAYER;CC03:C03;BB02:PLAYING_CARD;I:INT;C:PLAYING_CARD;
        R:RANK;S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
        FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE C03=0..1;
      TYPE PLAYER=1..2;
      TYPE PLAYING_CARD=0..51;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PILE2=SEQUENCE OF PLAYING_CARD  (AVCOMP=30 MAXCOMP=52);
      TYPE PILE1=(FRST11:E11;AA11:C11);
      TYPE C11=ARRAY D11 OF PLAYING_CARD;
      TYPE D11=1..52;
      TYPE E11=0..52;
      TYPE PLAYERS=ARRAY F09 OF INT;
      TYPE F09=0..5;
>REPS PILE2
CURRENTLY STOR=     912B
CHOICE FOR PILE2
  SEQ1             STOR=     950B
  SEQ2             STOR=    1274B
  SEQ3             STOR=    1591B
  SEQ5             STOR=    1274B
  SEQ6             STOR=     950B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SEQ1
```

```
      TYPE GLOBAL=(I12:INT;I11:INT;EQ07:LOGICAL;BB07:C03;
        EQ05:LOGICAL;BB05:PLAYER;CC03:C03;BB02:PLAYING_CARD;I:INT;
        C:PLAYING_CARD;R:RANK;S:SUIT;PLAYED:LOGICAL;P:PLAYER;
        FACE_DOWN:PILE2;FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS
      TYPE C03=0..1;
      TYPE PLAYER=1..2;
      TYPE PLAYING_CARD=0..51;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PILE2=(FRST12:E12;AA12:C12);
      TYPE C12=ARRAY D12 OF PLAYING_CARD;
      TYPE D12=1..52;
      TYPE E12=0..52;
      TYPE PILE1=(FRST11:E11;AA11:C11);
      TYPE PLAYERS=ARRAY F09 OF INT;
      TYPE E11=0..52;
      TYPE C11=ARRAY D11 OF PLAYING_CARD;
      TYPE F09=0..5;
      TYPE D11=1..52;
>REPS PLAYING_CARD
CURRENTLY STOR=     950B
CHOICE FOR PLAYING_CARD
  SUBREP           STOR=    3706B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBREP

      TYPE GLOBAL=(I12:INT;I11:INT;EQ07:LOGICAL;BB07:C03;
        EQ05:LOGICAL;BB05:PLAYER;CC03:C03;BB02:INT;I:INT;C:INT;
        R:RANK;S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
        FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE C03=0..1;
      TYPE PLAYER=1..2;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PILE2=(FRST12:E12;AA12:C12);
      TYPE PILE1=(FRST11:E11;AA11:C11);
      TYPE PLAYERS=ARRAY F09 OF INT;
      TYPE E12=0..52;
      TYPE C12=ARRAY D12 OF INT;
      TYPE E11=0..52;
      TYPE C11=ARRAY D11 OF INT;
      TYPE F09=0..5;
      TYPE D12=1..52;
      TYPE D11=1..52;
>REPS E11
CURRENTLY STOR=    3706B
CHOICE FOR E11
  SUBREP           STOR=    3732B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBREP

      TYPE GLOBAL=(I12:INT;I11:INT;EQ07:LOGICAL;BB07:C03;
        EQ05:LOGICAL;BB05:PLAYER;CC03:C03;BB02:INT;I:INT;C:INT;
        R:RANK;S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
        FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE C03=0..1;
      TYPE PLAYER=1..2;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PILE2=(FRST12:E12;AA12:C12);
      TYPE PILE1=(FRST11:INT;AA11:C11);
      TYPE PLAYERS=ARRAY F09 OF INT;
      TYPE E12=0..52;
      TYPE C12=ARRAY D12 OF INT;
      TYPE C11=ARRAY D11 OF INT;
      TYPE F09=0..5;
      TYPE D12=1..52;
      TYPE D11=1..52;
>REPS E12
CURRENTLY STOR=    3732B
CHOICE FOR E12
  SUBREP           STOR=    3753B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBREP
```

```
     TYPE GLOBAL=(I12:INT;I11:INT;EQ07:LOGICAL;BB07:C03;
       EQ05:LOGICAL;BB05:PLAYER;CC03:C03;BB02:INT;I:INT;C:INT;
       R:RANK;S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
       FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
     TYPE C03=0..1;
     TYPE PLAYER=1..2;
     TYPE RANK=1..13;
     TYPE SUIT=1..4;
     TYPE PILE2=(FRST12:INT;AA12:C12);
     TYPE PILE1=(FRST11:INT;AA11:C11);
     TYPE PLAYERS=ARRAY F09 OF INT;
     TYPE C12=ARRAY D12 OF INT;
     TYPE C11=ARRAY D11 OF INT;
     TYPE F09=0..5;
     TYPE D12=1..52;
     TYPE D11=1..52;
>REPS PILE1
CURRENTLY STOR=    3758B
CHOICE FOR PILE1
  SYNCP            STOR=    3758B
  SYN2             STOR=    3758B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SYN2

     TYPE GLOBAL=(I12:INT;I11:INT;EQ07:LOGICAL;BB07:C03;
       EQ05:LOGICAL;BB05:PLAYER;CC03:C03;BB02:INT;I:INT;C:INT;
       R:RANK;S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
       FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
     TYPE C03=0..1;
     TYPE PLAYER=1..2;
     TYPE RANK=1..13;
     TYPE SUIT=1..4;
     TYPE PILE2=(FRST12:INT;AA12:C12);
     TYPE PILE1=ARRAY E16 OF INT;
     TYPE E16=1..53;
     TYPE D11=1..52;
     TYPE PLAYERS=ARRAY F09 OF INT;
     TYPE C12=ARRAY D12 OF INT;
     TYPE F09=0..5;
     TYPE D12=1..52;
>REPS PILE2
CURRENTLY STOR=    3758B
CHOICE FOR PILE2
  SYNCP            STOR=    3758B
  SYN2             STOR=    3758B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SYN2

     TYPE GLOBAL=(I12:INT;I11:INT;EQ07:LOGICAL;BB07:C03;
       EQ05:LOGICAL;BB05:PLAYER;CC03:C03;BB02:INT;I:INT;C:INT;
       R:RANK;S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
       FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
     TYPE C03=0..1;
     TYPE PLAYER=1..2;
     TYPE RANK=1..13;
     TYPE SUIT=1..4;
     TYPE PILE2=ARRAY E17 OF INT;
     TYPE E17=1..53;
     TYPE D12=1..52;
     TYPE PILE1=ARRAY E16 OF INT;
     TYPE PLAYERS=ARRAY F09 OF INT;
     TYPE E16=1..53;
     TYPE F09=0..5;
>REPS C03
CURRENTLY STOR=    3758B
CHOICE FOR C03
  SUBREP           STOR=    3820B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBREP
```

```
      TYPE GLOBAL=(I12:INT;I11:INT;EQ07:LOGICAL;BB07:INT;
        EQ05:LOGICAL;BB05:PLAYER;CC03:INT;BB02:INT;I:INT;C:INT;
        R:RANK;S:SUIT;PLAYED:LOGICAL;P:PLAYER;FACE_DOWN:PILE2;
        FACE_UP:PILE1;PLAYER:PLAYER;PLAYERS:PLAYERS);
      TYPE PLAYER=1..2;
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PILE2=ARRAY E17 OF INT;
      TYPE PILE1=ARRAY E16 OF INT;
      TYPE PLAYERS=ARRAY F09 OF INT;
      TYPE E17=1..53;
      TYPE E16=1..53;
      TYPE F09=0..5;
>REPS PLAYER
CURRENTLY STOR=    3820B
CHOICE FOR PLAYER
  SUBREP          STOR=    3913B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBREP

      TYPE GLOBAL=(I12:INT;I11:INT;EQ07:LOGICAL;BB07:INT;
        EQ05:LOGICAL;BB05:INT;CC03:INT;BB02:INT;I:INT;C:INT;R:RANK;
        S:SUIT;PLAYED:LOGICAL;P:INT;FACE_DOWN:PILE2;FACE_UP:PILE1;
        PLAYER:INT;PLAYERS:PLAYERS);
      TYPE RANK=1..13;
      TYPE SUIT=1..4;
      TYPE PILE2=ARRAY E17 OF INT;
      TYPE PILE1=ARRAY E16 OF INT;
      TYPE PLAYERS=ARRAY F09 OF INT;
      TYPE E17=1..53;
      TYPE E16=1..53;
      TYPE F09=0..5;
>REPS RANK
CURRENTLY STOR=    3913B
CHOICE FOR RANK
  SUBREP          STOR=    3941B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBREP

      TYPE GLOBAL=(I12:INT;I11:INT;EQ07:LOGICAL;BB07:INT;
        EQ05:LOGICAL;BB05:INT;CC03:INT;BB02:INT;I:INT;C:INT;R:INT;
        S:SUIT;PLAYED:LOGICAL;P:INT;FACE_DOWN:PILE2;FACE_UP:PILE1;
        PLAYER:INT;PLAYERS:PLAYERS);
      TYPE SUIT=1..4;
      TYPE PILE2=ARRAY E17 OF INT;
      TYPE PILE1=ARRAY E16 OF INT;
      TYPE PLAYERS=ARRAY F09 OF INT;
      TYPE E17=1..53;
      TYPE E16=1..53;
      TYPE F09=0..5;
>REPS SUIT
CURRENTLY STOR=    3941B
CHOICE FOR SUIT
  SUBREP          STOR=    3971B
TO IMPLEMENT REP, GIVE CHOSEN REP NAME
>SUBREP

      TYPE GLOBAL=(I12:INT;I11:INT;EQ07:LOGICAL;BB07:INT;
        EQ05:LOGICAL;BB05:INT;CC03:INT;BB02:INT;I:INT;C:INT;R:INT;
        S:INT;PLAYED:LOGICAL;P:INT;FACE_DOWN:PILE2;FACE_UP:PILE1;
        PLAYER:INT;PLAYERS:PLAYERS);
      TYPE PILE2=ARRAY E17 OF INT;
      TYPE PILE1=ARRAY E16 OF INT;
      TYPE PLAYERS=ARRAY F09 OF INT;
      TYPE E17=1..53;
      TYPE E16=1..53;
      TYPE F09=0..5;
>EVAL
CURRENTLY STOR=    3971B       TIME=   5.58'+04
```

```
>PRINT
TYPE GLOBAL=(I12:INT;I11:INT;EQ07:LOGICAL;BB07:INT;
      EQ05:LOGICAL;BB05:INT;CC03:INT;BB02:INT;I:INT;C:INT;R:INT;
      S:INT;PLAYED:LOGICAL;P:INT;FACE_DOWN:PILE2;FACE_UP:PILE1;
      PLAYER:INT;PLAYERS:PLAYERS);
TYPE PILE2=ARRAY E17 OF INT;
TYPE PILE1=ARRAY E16 OF INT;
TYPE PLAYERS=ARRAY F09 OF INT;
TYPE E17=1..53;
TYPE E16=1..53;
TYPE F09=0..5;
  BEGIN
  G.FACE_UP.INDEX(53):=(0);
  G.FACE_DOWN.INDEX(53):=(0);
  FOR G.BB05:1..2 DO
    BEGIN
    G.PLAYERS.INDEX((G@'BB05+3)):=(0);
    FOR G.BB07:0..1 DO
      G.PLAYERS.INDEX(((G@'BB07*2)+(G@'BB05-1))):=(0);
    END;
  FOR G.I:0..51 DO
    BEGIN
    G.FACE_DOWN.INDEX(53):=((G@'FACE_DOWN'INDEX(53)+1));
    G.FACE_DOWN.INDEX(G@'FACE_DOWN'INDEX(53)):=
     ((((((G@'I REM 13)+1)-1)*4)+(((G@'I DIV 13)+1)-1)));
    END;
  FOR G.P:1..2 DO
    FOR G.I:1..7 DO
      BEGIN
      G.PLAYERS.INDEX((((G@'FACE_DOWN'INDEX(G@'FACE_DOWN'INDEX(53))
        DIV 32)*2)+(G@'P-1))):SETBIT((G@'FACE_DOWN'INDEX(G@
        'FACE_DOWN'INDEX(53)) REM 32),  TRUE);
      G.PLAYERS.INDEX((G@'P+3)):=((G@'PLAYERS'INDEX((G@'P+3))+1));
      G.FACE_DOWN.INDEX(53):=((G@'FACE_DOWN'INDEX(53)-1));
      END;
  G.FACE_UP.INDEX(53):=((G@'FACE_UP'INDEX(53)+1));
  G.FACE_UP.INDEX(G@'FACE_UP'INDEX(53)):=
    (G@'FACE_DOWN'INDEX(G@'FACE_DOWN'INDEX(53)));
  G.FACE_DOWN.INDEX(53):=((G@'FACE_DOWN'INDEX(53)-1));
  G.PLAYER:=(1);
  WHILE ¬(G@'PLAYERS'INDEX(4)=0 OR G@'PLAYERS'INDEX(5)=0) DO
    BEGIN
    G.PLAYED:=( FALSE);
    G.S:=(1);
    WRITE(((G@'FACE_UP'INDEX(G@'FACE_UP'INDEX(53)) DIV 4)+1));
    WRITE(((G@'FACE_UP'INDEX(G@'FACE_UP'INDEX(53)) REM 4)+1));
    WHILE (¬G@'PLAYED AND G@'S<=4) DO
      BEGIN
      G.C:=((((((G@'FACE_UP'INDEX(G@'FACE_UP'INDEX(53)) DIV 4)
        +1)-1)*4)+(G@'S-1)));
      IF G@'PLAYERS'INDEX((((G@'C DIV 32)*2)+(G@'PLAYER-1)))
          'BIT((G@'C REM 32)) THEN
        BEGIN
        G.FACE_UP.INDEX(53):=((G@'FACE_UP'INDEX(53)+1));
        G.FACE_UP.INDEX(G@'FACE_UP'INDEX(53)):=(G@'C);
        G.PLAYERS.INDEX((((G@'C DIV 32)*2)+(G@'PLAYER-1)))
          :SETBIT((G@'C REM 32), FALSE);
        G.PLAYERS.INDEX((G@'PLAYER+3)):=
          ((G@'PLAYERS'INDEX((G@'PLAYER+3))-1));
        G.PLAYED:=(  TRUE);
        END
      ELSE
        G.S:=((G@'S+1));
      END;
    G.R:=(1);
    WHILE (¬G@'PLAYED AND G@'R<=13) DO
      BEGIN
      G.C:=((((G@'R-1)*4)+(((G@'FACE_UP'INDEX(G@'FACE_UP'INDEX(53))
              REM 4)+1)-1))));
      IF G@'PLAYERS'INDEX((((G@'C DIV 32)*2)+(G@'PLAYER-1)))
           'BIT((G@'C REM 32)) THEN
        BEGIN
        G.FACE_UP.INDEX(53):=((G@'FACE_UP'INDEX(53)+1));
        G.FACE_UP.INDEX(G@'FACE_UP'INDEX(53)):=(G@'C);
        G.PLAYERS.INDEX((((G@'C DIV 32)*2)+(G@'PLAYER-1)))
          :SETBIT((G@'C REM 32), FALSE);
        G.PLAYERS.INDEX((G@'PLAYER+3)):=
          ((G@'PLAYERS'INDEX((G@'PLAYER+3))-1));
        G.PLAYED:=(  TRUE);
        END
      ELSE
        G.R:=((G@'R+1));
      END;
  IF ¬G@'PLAYED THEN
    BEGIN
    WRITE(G@'PLAYER);
    WRITE(((G@'FACE_DOWN'INDEX(G@'FACE_DOWN'INDEX(53)) DIV 4)+1));
    WRITE(((G@'FACE_DOWN'INDEX(G@'FACE_DOWN'INDEX(53)) REM 4)+1));
    G.PLAYERS.INDEX((((G@'FACE_DOWN'INDEX(G@'FACE_DOWN'INDEX(53))
      DIV 32)*2)+(G@'PLAYER-1))):SETBIT((G@'FACE_DOWN'INDEX(G@
        'FACE_DOWN'INDEX(53)) REM 32),  TRUE);
    G.PLAYERS.INDEX((G@'PLAYER+3)):=
      ((G@'PLAYERS'INDEX((G@'PLAYER+3))+1));
    G.FACE_DOWN.INDEX(53):=((G@'FACE_DOWN'INDEX(53)-1));
    END;
  G.PLAYER:=(((G@'PLAYER REM 2)+1));
  END;
WRITE(G@'PLAYERS'INDEX(4)=0);
END
```

The final action in the above session was to print the concrete form of the program in the system notation.

As was done for the Birthdays program in section 5.4.1, the final program for the Card Game problem has been transliterated into AlgolW to give the program shown on the following page. This program can then be compiled and executed as normal.

```
BEGIN INTEGER I12,I11,CC03,BB02,C,R,S,PLAYER;
LOGICAL EQ07,EQ05,PLAYED;
INTEGER ARRAY FACE_DOWN(1::53);
INTEGER ARRAY FACE_UP(1::53);
INTEGER ARRAY PLAYERS(0::5);
LOGICAL PROCEDURE BIT(INTEGER VALUE I,BITNUM);
  ((BITSTRING(I) SHL BITNUM) AND #8000) = #8000;
PROCEDURE SETBIT(INTEGER VALUE RESULT I; INTEGER VALUE BITNUM;
                 LOGICAL VALUE ON);
   IF ON THEN I:=NUMBER(BITSTRING(I) OR (#8000 SHR BITNUM))
         ELSE I:=NUMBER(BITSTRING(I) AND ¬(#8000 SHR BITNUM));
   BEGIN
   FACE_UP(53):=(0);
   FACE_DOWN(53):=(0);
   FOR BB05:=1 UNTIL 2 DO
     BEGIN
     PLAYERS((BB05+3)):=(0);
     FOR BB07:=0 UNTIL 1 DO
       PLAYERS(((BB07*2)+(BB05-1))):=(0);
     END;
   FOR I:=1 UNTIL 52 DO
     BEGIN
     FACE_DOWN(53):=((FACE_DOWN(53)+1));
     FACE_DOWN(FACE_DOWN(53)):=
      ((((((I REM 13)+1)-1)*4)+(((I DIV 13)+1)-1)));
     END;
   FOR P:=1 UNTIL 2 DO
     FOR I:=1 UNTIL 7 DO
       BEGIN
       SETBIT(PLAYERS((((FACE_DOWN(FACE_DOWN(53))
         DIV 32)*2)+(P-1))),(FACE_DOWN(
         FACE_DOWN(53)) REM 32),  TRUE);
       PLAYERS((P+3)):=((PLAYERS((P+3))+1));
       FACE_DOWN(53):=((FACE_DOWN(53)-1));
       END;
   FACE_UP(53):=((FACE_UP(53)+1));
   FACE_UP(FACE_UP(53)):=
     (FACE_DOWN(FACE_DOWN(53)));
   FACE_DOWN(53):=((FACE_DOWN(53)-1));
   PLAYER:=(1);
   WHILE ¬(PLAYERS(4)=0 OR PLAYERS(5)=0) DO
     BEGIN
     PLAYED:=( FALSE);
     S:=(1);
     WRITE(((FACE_UP(FACE_UP(53)) DIV 4)+1));
     WRITEON(((FACE_UP(FACE_UP(53)) REM 4)+1));
     WHILE (¬PLAYED AND S<=4) DO
       BEGIN
       C:=((((((FACE_UP(FACE_UP(53)) DIV 4)
         +1)-1)*4)+(S-1)));
       IF BIT(PLAYERS((((C DIV 32)*2)+(PLAYER-1)))
          ,(C REM 32)) THEN
         BEGIN
         FACE_UP(53):=((FACE_UP(53)+1));
         FACE_UP(FACE_UP(53)):=(C);
         SETBIT(PLAYERS((((C DIV 32)*2)+(PLAYER-1)))
           ,(C REM 32), FALSE);
         PLAYERS((PLAYER+3)):=
           ((PLAYERS((PLAYER+3))-1));
         PLAYED:=(  TRUE);
         END
       ELSE
         S:=((S+1));
       END;
     R:=(1);
     WHILE (¬PLAYED AND R<=13) DO
       BEGIN
       C:=((((R-1)*4)+(((FACE_UP(FACE_UP(53))
             REM 4)+1)-1)));
       IF BIT(PLAYERS((((C DIV 32)*2)+(PLAYER-1)))
            ,(C REM 32)) THEN
         BEGIN
         FACE_UP(53):=((FACE_UP(53)+1));
         FACE_UP(FACE_UP(53)):=(C);
         SETBIT(PLAYERS((((C DIV 32)*2)+(PLAYER-1)))
           ,(C REM 32), FALSE);
         PLAYERS((PLAYER+3)):=
           ((PLAYERS((PLAYER+3))-1));
         PLAYED:=(  TRUE);
         END
       ELSE
         R:=((R+1));
       END;
      IF ¬PLAYED THEN
        BEGIN
        WRITE(PLAYER," PICKS UP ");
        WRITEON(((FACE_DOWN(FACE_DOWN(53)) DIV 4)+1));
        WRITEON(((FACE_DOWN(FACE_DOWN(53)) REM 4)+1));
        SETBIT(PLAYERS(((FACE_DOWN(FACE_DOWN(53))
          DIV 32)*2)+(PLAYER-1))),(FACE_DOWN(
            FACE_DOWN(53)) REM 32),  TRUE);
        PLAYERS((PLAYER+3)):=
          ((PLAYERS((PLAYER+3))+1));
        FACE_DOWN(53):=((FACE_DOWN(53)-1));
        END;
      PLAYER:=(((PLAYER REM 2)+1));
      END;
    WRITE(PLAYERS(4)=0," PLAYER 1 WINS");
    END
  END.
```

REFERENCES

[1] Arsac, J.J.
'Program Transforms as a Programming Tool'.
Univ. Pierre et Marie Curie, Inst. de Programmation Report 1976.

[2] Balzer, R.M.
'Dataless Programming'.
Proc. AFIPS 1967 FJCC. pp 535-544.

[3] Balzer, R. , Goldman, N. & Wile, D.
'On the Transformational Implementation Approach to Programming'.
Proc. 2nd Int.Conf. on Software Engineering, San Francisco,
Oct. 1976; pp 337-344.

[4] Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, E. & Strachey, C.
'The Main Features of CPL'.
Computer Journal, 6. 1963. pp 134-143.

[5] Barron, D.W.
'An Introduction to the Study of Programming Languages'.
Cambridge University Press 1977.

[6] Burstall, R.M. & Darlington, J.
'A Transformation System for Developing Recursive Programs'.
JACM 24,1. Jan. 1977. pp 44-67.

[7] Cheatham, T.E.Jnr.
'The Recent Evolution of Programming Languages'.
Proc. IFIP Congress, Aug. 1971. pp I/118-I/134.

[8] Cheatham, T.E.Jnr. & Wegbreit, B.
'A Laboratory for the Study of Automatic Programming'.
Proc. AFIPS 1972 SJCC. pp 11-21.

[9] Cohen, J. & Zuckerman, C.
'Two Languages for Estimating Program Efficiency'.
CACM 17,6. June 1974. pp 301-308.

[10] Dahl, O-J. , Myhrhaug, B. & Nygaard, K.
'Simula Common Base Language'.
Norwegian Computing Centre 1970.

[11] Darlington, J. & Burstall, R.M.
'A System which Automatically Improves Programs'.
Proc. 3rd Int. Conf. on A.I., Stanford 1973. pp 479-485.

[12] Dijkstra, E.W.
'Notes on Structured Programming'.
In Dahl,Dijkstra,Hoare 'Structured Programming'. Academic Press 1972.

[13] Dijkstra, E.W.
'The Humble Programmer'.
CACM 15,10. Oct. 1972. pp 859-866.

[14] Dijkstra, E.W.
'A Discipline of Programming'.
Academic Press 1976.

[15] D'Imperio, M.E.
'Data Structures and their Representation in Storage'.
ARAP 5, 1969. pp 1-75.

[16] Earley, J.
'Relational Level Data Structures for Programming Languages'.
Acta Informatica 2. 1973. pp 293-309.

[17] Feldman, J.A. & Rovner, P.D.
'An Algol-Based Associative Language'.
CACM 12,8. Aug. 1969. pp 439-449.

[18] Feldman, J.A. , Low, J.R. , Swinehart, D.C. & Taylor, R.H.
'Recent Developments in SAIL - an Algol-based language for A.I.'.
Proc. AFIPS 1972 FJCC. pp 1193-1202.

[19] Galler, B.A. & Perlis, A.J.
'A Proposal for Definitions in Algol'.
CACM 10,4. April 1967. pp 204-219.

[20] Gerhart, S.L.
'Correctness-Preserving Program Transformations'.
Proc. 2nd Symp. on Principles of Prog.Langs., Palo Alto, Jan. 1975. pp 54-66.

[21] Gerhart, S.L.
'Knowledge about Programs: A Model and Case Study'.
Proc. Int. Conf. on Reliable Software, Los Angeles, April 1975. pp 88-95.

[22] Gotlieb, C.C. & Tompa, F.W.
'Choosing a Storage Schema'.
Acta Informatica 3, 1974. pp 297-319.

[23] Henderson, P. & Snowdon, R.A.
'A Tool for Structured Program Development'.
Proc. IFIP Congress, Stockholm. Aug. 1974. pp 204-207.

[24] Henderson,P., Snowdon,R.A., Gorrie,J.D. & King,I.I.
'The TOPD System'.
Univ. of Newcastle-upon-Tyne Technical Report 77. Sept. 1975.

[25] Hoare, C.A.R.
'Notes on Data Structuring'.
In Dahl,Dijkstra,Hoare 'Structured Programming'. Academic Press 1972.

[26] Hoare, C.A.R.
'Proof of Correctness of Data Representations'.
Acta Informatica 1, 1972. pp 271-281.

[27] Hoare, C.A.R.
'Hints on Programming Language Design'.
Stanford Univ. Report STAN-CS-73-403. Dec. 1973.

[28] Jensen, K. & Wirth, N.
'PASCAL User Manual and Report'.
Lecture Notes in Comp. Sci. 18. Springer-Verlag 1974.

[29] Kant, E.
'The Selection of Efficient Implementations for a High-Level Language'.
SIGPLAN Notices 12,8. Aug. 1977. pp 140-146.

[30] Kibler, D.F., Neighbors, J.M. & Standish, T.A.
'Program Manipulation via an Efficient Production System'.
SIGPLAN Notices 12,8. Aug. 1977. pp 163-173.

[31] Knuth, D.E.
'Mathematical Analysis of Algorithms'.
Proc. IFIP Congress. Aug. 1971. pp I/135-I/143.

[32] Knuth, D.E.
'Structured Programming with 'goto' Statements'.
Comp. Surveys 6,4. 1974. pp 261-301.

[33] Liskov, B.
'A Note on CLU'.
MIT Computation Structures Group Memo 112-1. Nov. 1974.

[34] Liskov, B., Snyder, A., Atkinson, R. & Schaffert, C.
'Abstraction Mechanisms in CLU'.
CACM 20,8. Aug. 1977. pp564-576.

[35] Liskov, B. & Zilles, S.
'Programming with Abstract Data Types'.
SIGPLAN Notices 9,4. April 1974. pp 50-59.

[36] Liskov, B. & Zilles, S.
'Specification Techniques for Data Abstractions'.
IEEE Trans. on Software Engineering SE-1,1. March 1975. pp 7-19.

[37] Lomet, D.B.
'Objects and Values: The Basis of a Storage Model for Procedural Languages'.
IBM J. Res.& Dev. 20,2. March 1976. pp 157-167.

[38] Loveman, D.B.
'Program Improvement by Source to Source Transformation'.
JACM 24,1. Jan. 1977. pp 121-145.

[39] Low, J.R.
'Automatic Coding: Choice of Data Structure'.
Stanford Univ. Report STAN-CS-74-452. Aug. 1974.

[40] Low, J. & Rovner, P.
'Techniques for the Automatic Selection of Data Structures'.
Proc. 3rd ACM Symp. on Principles of Prog. Langs., Atlanta, Jan. 1976. pp 58-67.

[41] McCuskey, W.A.
'On Automatic Design of Data Organisation'.
Proc. AFIPS 1970 FJCC. pp 187-199.

[42] Mealy, G.H.
'Another Look at Data'.
Proc. AFIPS 1967 FJCC. pp 525-534.

[43] Middleton, A.G.
'A Structured Model of Programs for Analysing Time/Storage Trade-Offs'.
SIGPLAN Notices 9,9. Sept. 1974. pp 18-28.

[44] Naur, P. et.al.
'Revised Report on the Algorithmic Language Algol 60'.
CACM 6,1. Jan. 1963. pp 1-17.

[45] Rosenschein, S.J. & Katz, S.M.
'Selection of Representations for Data Structures'.
SIGPLAN Notices 12,8. Aug. 1977. pp 147-154.

[46] Rovner, P.D.
'Automatic Representation Selection for Associative Data Structures'.
Rochester Univ. Report TR10. Sept. 1976.

[47] Schwartz, J.T.
'Automatic and Semiautomatic Optimisation of SETL'.
SIGPLAN Notices 9,4. April 1974. pp 43-49.

[48] Schwartz, J.T.
'Automatic Data Structure Choice in a Language of Very High Level'.
CACM 18,12. Dec. 1975. pp 722-728.

[49] Standish, T.A., Kibler, D.F. & Neighbors, J.M.
'Improving and Refining Programs by Program Manipulation'.
Proc. ACM Conf., Houston, Oct. 1976. pp 509-516.

[50] Strachey, C.
'Towards a Formal Semantics'.
In T.B.Steel 'Formal Language Description Languages', North-Holland 1966. pp 198-220.

[51] Tompa, F.W.
'Evaluating the Efficiency of Storage Structures'.
Univ. of Waterloo Report CS-75-16. May 1975.

[52] Walk, K.
'Modelling of Storage Properties of Higher-Level Languages'.
Int.J. Comp.& Inf. Sci. 2,1. March 1973. pp 1-24.

[53] Wegbreit, B.
'Mechanical Program Analysis'.
CACM 18,9. Sept. 1975. pp 528-539.

[54] Wegbreit, B.
'Goal-Directed Program Transformation'.
Proc. 3rd Symp. on Principles of Prog. Langs., Atlanta,
Jan. 1976. pp 153-170.

[55] Weinberg, G.M.
'The Psychology of Computer Programming'.
Van Nostrand Reinhold Co. 1971.

[56] Wichmann, B.A.
'Algol 60 Compilation and Assessment'.
Academic Press 1973.

[57] Wijngaarden, A.van. et.al.
'Report on the Algorithmic Language Algol 68'.
Numerische Mathematik 14, 1969. pp 79-218.

[58] Wirth, N. & Hoare, C.A.R.
'A Contribution to the Development of Algol'.
CACM 9,6. June 1966. pp 413-432.

[59] Wulf,W.A., Russell,D.B. & Habermann,A.N.
'BLISS: a Language for Systems Programming'.
CACM 14,12. Dec. 1971. pp 780-790.

[60] Wulf,W.A., London,R.L. & Shaw,M.
'Abstraction and Verification in ALPHARD: Introduction
to Language & Methodology'.
USC-ISI Report ISI/RR-76-46. June 1976.