ORTHOGONAL PERSISTENCE

An abstract representation of persistent storage in Algol-like languages

William Paul Cockshott

Ph D University of Edinburgh 1982

CONTENTS

Abstract

The abstract representation of persistent storage in conventional programming languages is examined and criticised for being modeled upon serial input/output devices. The introduction of database models into programming language is reviewed and is found to introduce non-orthogonal elements into language design. It is argued that persistence should be an orthogonal feature of data and that its duration should be determined by a combination of scope rules and heap reachability criteria. The architecture of a distributed computing system to implement this model is presented. The possibility of a system for the cannonical representation and storage of persistent data from Algol-68 and Pascal is examined and rejected. The design and implementation of a prototype configuration supporting orthogonal persistence in the language S-algol is described. Algorithms for the maintenance of the persistent heap in the prototype are evaluated. The weaknesses of the prototype language are criticised and an extended language NEPAL proposed to overcome these. The implications of orthogonal data persistence for scope rules, garbage collection, and concurrency control are examined.

Preface

This thesis is the result of work carried out as part of the Data Curator Project: a Science Research Council funded investigation into databases and programming languages.

Ken Chisholm and the author were employed on this project, working under the supervision of Dr M. P. Atkinson. Within the project Ken Chisholm worked almost exclusively on the Chunk Management System mentioned in Chapter 5. The other software components described in this document were the work of the author with the exception of the 3220 and Vax scode interpreters, parts of which were produced by Richard Marshall a vacation student temporarily employed on the project and Paul Mclellan. The model of orthogonal persistence outlined in Chapter 3 was arrived at by the author on the basis of a critical examination of prior proposals by Atkinson. The proposals for system architecture in Chapter 4 including the initial version of the PIDLAM idea derive in large part from proposals by Atkinson. The examination of and eventual rejection of the idea of a cannonical type representation was the work of the author in conjunction with B. Monahan. The heap algorithms described in Chapter 6 were the work of the author.

The choice of the language S-algol as the most suitable for extension and its subsequent extension into NEPAL were arrived at by the author in conjunction with Atkinson. The investigation of the relationship between scope rules and garbage collection was the work of the author.

Chapter 1

Programming Languages and Persistent Data

This is a defence of the thesis that it is technically feasible and pragmatically desirable to add data persistence as an orthogonal feature to a certain class of algorithmic languages.

By persistence of data used in a program, I mean that the data should remain available to the same or other programs over a period during which the computer or computers on which the data is held may have run many programs and may have been switched off.

The history of programming languages is characterised by a markedly uneven development of their facilities for handling temporary and persistent data.

The temporary data has been provided with sucessively more sophisticated structuring facilities. Even the most basic of algorithmic languages provide facilities for the symbolic naming of data items and for the construction of vectors. As language design has progressed one has had the notion of type applied to data items. Initially one had a limited set of predefined types supported by the language: integers, reals, logical or boolean values. These facilities were available in languages of the early 1960's such as Fortran or Algol60, or even the autocodes of the late 50's.

The concept was then generalised to include programmer defined data types formed by a group of type construction operators such as enumeration, set definition, cartesian composition, referencing and set union. These facilities became available by the late 1960's with such languages as Algol W, Algol 68 and Pascal. [1] [2] [3]

Finally one has seen the development of the notion of an abstract data type defined in terms of the operations upon members of that type rather than in terms of its representation. This appeared first in Simula, [4] was extensively experimented on during the 1970's, e.g. Mesa, CLU, EUCLID [5] [6] [7] and is likely to become widely accepted with the programming language Ada [8]

The language definitions which brought us these facilities are, however, marked by a silence that is at first inaudible. They assume, but do not explicitly say, that the data for whose structuring they provide, is to be held in the random access memory of a computer. The data world of the programming language is well ordered but evanescent. Within it data stands in its classes or orders, governed by the rule of law, with each class kept in its place and prohibited by sumptuary provision from impersonating its superiors. But like all mundane orders it eventually meets its end, exits from its final block and is no more. Then its territories are put on the free list to be reallocated and the memories of its former variables overwritten or reinitialised.

Data can only persist beyond the pale, on the outside, in the world of files where, it is said, you can live for, if not ever, at least a day. Dealings with the world beyond take place by input and output operations. But these are risky

undertakings. For all its promises of immortality the outside is a chaotic, anarchic place. It acknowledges no class or rank higher than the byte and no law other than sequence. Integers and reals alike are reduced to sequences of bytes; while records or enumerated types, let alone abstract types cannot enter or leave.

Input data poses even more problems. It enters without a pedigree; how is one to know to which class it is to be assigned. One must set up elaborate immigration procedures on input to sort out the integers from the strings and reals, to regiment these together into higher order types.

Inside and Outside

This sharp division between the world within and the world outside the program is a relic of an earlier stage of computing. It is the reflection within programming languages of an old diagram, familiar to us all, that divided computers into three parts: a cpu, a memory and peripherals (Fig 1.1).

The cpu, it was explained, did calculations and interpreted instructions which it fetched from memory to process inputs from peripherals and the results were sent back out to peripherals. And what were these peripherals?

First of course there were the tape drives, symbols of the computer age. Then there were the card readers, printers, teletypes and discs. But for the programmer they were all peripherals, and their details peripheral to her concerns. Her programs dealt with input and output streams, their material incarnations as devices was nothing to do with the program and could be relegated to the operating system and the JCL. The operating system presented a device independent view of peripherals. To change the devices that a program used one had merely to assign different devices to the streams or channels used by the program. But this freedom was bought at a cost that we are still paying. All devices must be made to look alike. All must be reduced to the lowest common denominator: input devices end up looking like a paper tape reader, output devices like a teletype. The operations on them are reduced to the barest essentials: on input, look at the character under the read head or step the tape on, for output just print a character.

Time has passed, the technology of storage devices has advanced and now we have a new diagram. It is called the storage hierarchy.

It is in the shape of a pyramid (Fig 1.2). [9] At the base are the disc drives. Upon them stands the MOS main memory. Above this is the bipolar cache and at the apex sit the cpu registers. All of these, we are told, are storage devices. What distinguishes them is just speed. All of them are randomly accessible and all may be read from or written to. The older generation of operating systems was concerned to make all peripherals look like paper tape. The new virtual memory systems make all store look like main memory, but like a much bigger and, if the cache works, faster main memory.

From the viewpoint of the operating systems engineer this is true, but the poor programmers are left with languages of the previous era. Yes they can now declare arrays of a million elements with carefree abandon and the operating

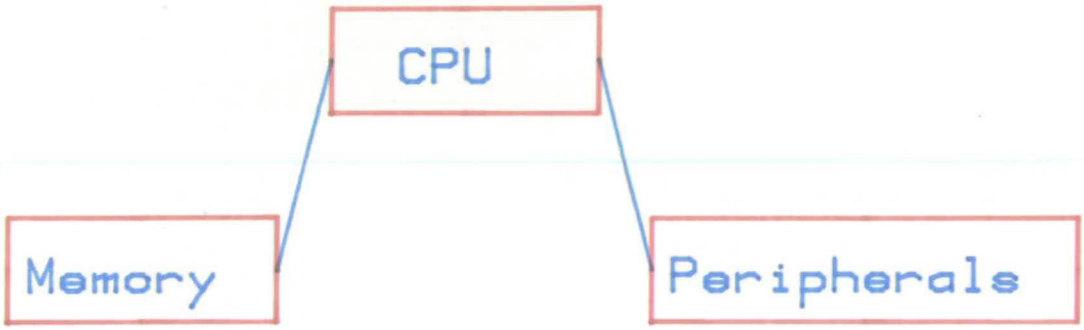

Fig 1.1 We still think like this in programming languages

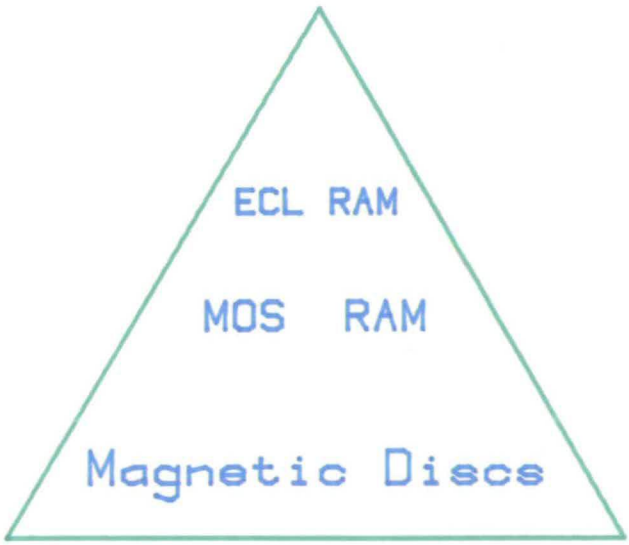

Fig 1.2 The storage hierarchy

system will take care of mapping these onto the appropriate level of the storage hierarchy on demand. But from the standpoint of the language, an array is an array wherever it resides during its still short life. It is still thrown away at the end of the program. What is left at the end of a program is just what has been written out to a file. Nowadays the file will be on disk but, in basic conception, it still looks like a serial peripheral device to the program.

We have advanced somewhat. We are no longer forced to look at discs as if they were paper tape, we can also make them look like RAM. From one point of view they are a set of files: securely locked cupboards with neatly rolled, carefully labeled rolls of paper tape. The operating system acts as diligent operator who will fetch named tapes from the cupboard and put them on the reader. From the other point of view they are a giant scratchpad RAM and the operating system is a rather slow address decoding chip.

In the EMAS system, [10] this view reaches an apex with all files being mapped onto random access memory. Memory mapped files are a very significant advance on ordinary files. They would seem to provide an answer to the problem of making a program's run time data structures persist. Under EMAS a file can be mapped onto an array which is used in the conventional fashion within a program, and then at the end of the program the data in this is preserved in the file. There are, however, two fundamental weaknesses to memory mapped files as implemented on EMAS.

1. The programming language support for memory mapped files is very weak. There is no type checking of the data in the mapped file. The use of such files relies entirely upon the programmer to get her type declarations for the memory mapped files right. All programs which use a file must have the same view of it, and there is no check that when one maps a file it is of the appropriate type. Furthermore, none of the languages provided under the operating system use these files in conjunction with a heap. This leads to needless multiplication of effort, with every program needing its own ad hoc storage allocation mechanism.

2. Memory mapped files are unsafe. Data in a memory mapped file is corruptible in the event of program or machine malfunction. Either of these can cause a database held in such a file to be irretrievably corrupted. In other words EMAS memory mapped files do not support the notion of atomic transaction. By atomic transaction (henceforth transaction) I mean an all or nothing modification to a body of data. If a transaction succeeds all the changes required will have taken place, if it fails, none will have. In a conventional file system, transactions are frequently done by creating a new version of a file whenever you write to it. If used with memory mapped files, this would provide a secure, if space consuming, way of implementing transactions on program data structures. The EMAS filing system does not provide file versions for mapped files, so that this approach to providing transactions is unimplementable.

Although the memory mapped file approach seems a promising line of development given a little more work, it has not been part of the mainstream of

computing culture. This was concerned to let discs appear as what they are: random access persistent storage devices.

Another World Another History

Programming language designers seem to have been too busy elsewhere to give much thought to how to represent persistent storage devices within programming languages. But this did not deter applications programmers from attempting the task, and in the process giving rise to a new world: the world of database systems. This is a world with its own history, culture and traditions, but it is a history and culture that has few points of contact with that of programming languages. Database systems and programming languages have both evolved in response to constraints, some of the constraints and problems have been the same, others different, giving rise to an evolution that was at times divergent and at others convergent. Programming languages, or more properly, algorithmic programming languages have evolved as a means of precisely specifying algorithms for the transformation and manipulation of externally supplied data. Their data structuring facilities have developed as a means of handling the data structures needed in algorithms. Algorithmic languages are a means of programming with fast random access memory, data base systems are a means of programming with discs. In keeping with the characteristics of the device, database systems are not so much concerned with the transformation of data, but with its preservation, organisation and access.

In contrast to MOS RAM, discs are large in capacity, have a slow cycle time and if we consider disc blocks to be analogous to words, a large word length and of course the information in them does not vanish when you turn the power off. Each of these features has had its effect on the divergence of the evolution of database systems from that of programming languages.

The large capacity has meant that the amount of data to be handled is much greater than in algorithmic languages. This has made the provision of automatic storage allocation necessary. All algorithmic languages also have some form of automatic storage allocation, if only for named variables, but in general this relies upon knowing beforehand the order in which store will be requested and released. Only a few languages routinely provide a full heap regime with automatic allocation and recovery of objects in any arbitrary order. This, however, is always provided by database systems.

The long word length and slow cycle time of discs in combination with the large amount of data stored has led to an emphasis on the provision of means of efficiently indexing large populations of data using sparse keys. The non-volatile nature of disc storage has meant that database systems are primarily concerned with the maintenance rather than the manipulation of data. In turn, this has led to a concern for robustness and security of data storage and the notion of transaction and recovery.

Despite this concern with the maintenance and organisation of stored data, this data has still in most applications to be manipulated and transformed. Given the pre-existence of programming languages optimised for the task of manipulating and transforming data, an obvious way of dealing with this is to interface the database system to some existing programming language. It is worth giving a

short outline of the phases through which this process of interfacing programming languages to database systems has passed. As with most technological developments its history is only partially sequential. The earlier phases of the technology continue in use well after newer ones have been invented.

The earliest facility provided to programming languages for handling databases stored on disc was the direct access file. In essence this gives the programmer a virtual disc drive. It allows only the primitive disc operations of reading and writing numbered blocks. The only difference between this and directly operating on the disc itself is that the operating system restricts these operations to a subset of the total disc. To the programmer the facility looks like either a modified input or output statement or a subroutine, which when given a number and a buffer transfer the contents of the numbered disk block to or from the buffer. As a programming technique this is about on a par with programming a single accumulator machine with an assembler that does not even provide symbolic labels. The programmers must provide their own rules for allocating storage, and all dereferencing and assignment must be done with explicit operations. Programming in this way is tedious and prone to errors of the most elementary kind: getting addresses mixed up, failure to carry out the necessary level of dereferencing. Although the organisation of data within a disc block may be handled cleanly if the host language supports records, there is no type protection. There is no guarantee that when you read in a disc block that it will contain the type of record that you want. As against these problems it does have the advantage that it is very powerful. You have lost none of the underlying machine capabilities. All more sophisticated database systems must ultimately be implemented in terms of operations at this level.

At a level up from direct access files one has subroutine access to record management software. I am thinking here of systems which allow the reading and writing of variable sized records, and which handle the automatic allocation of records and the recovery of space when these are deleted. In other words a record based disc store. Whilst this relieves the programmer of the problem of storage allocation, a number of the other headaches remain. There is still no type protection. When following a chain of pointers, one must still carry out explicit fetches to do the dereferencing. Programming at this level brings all the problems associated with working directly upon the representations of data types. Knowledge of these representations becomes spread throughout the algorithms that work on the data. This is bad enough within a single program, but when dealing with whole groups of programs written by different authors, acting on common persistent data structures, the situation becomes unmanageable. Any change to the data structure has implications throughout the whole set of programs. Fear of the consequences of such changes leads to data structures becoming frozen.

By the early 1970's, awareness of these problems led to the development of database systems that incorporated data descriptions on disc as a database schema. Proposals for the two currently leading database models, the network and relational were formulated [11] [12] These enabled the definition of data structures that were accessed via certain well defined operations that effectively enforced type protection. In addition, more sophisticated access paths were incorporated, allowing the application programmer to avoid having to do explicit pointer following. With the development of database systems with schemata,

automatic storage control and sophisticated access methods, database systems had advanced to a level of data handling that was perhaps analogous to that reached by algorithmic languages of the early 1970's. Indeed if one considers relations or CODASYL sets to be type constructors, the level reached was somewhat higher.

The first method by which access was provided to such systems was by subroutine call. This was in keeping with the usual method by which the facilities in existing programming languages had been extended. The continued popularity of Fortran, for instance, is testimony to the effectiveness of the enrichment of a language by the use of subroutine libraries. In the database context, the applications programmer would write a call on a subroutine in the DBMS library requesting some kind of retrieval or update operation and passing the address of a buffer into which or from which the data was to come. While this technique of enrichment has been successful in a number of areas of computing, graphical or mathematical subroutine libraries spring immediately to mind, there were a number of reasons why it was less effective in the case of database systems.

1. In database systems there are a larger number of potential error conditions than in many other applications. Retrieval requests may often be issued for which there is no target record in the database. Such conditions must be caught and handled by the applications program. A subroutine call interface makes the communication of error status information difficult. The information must be passed back as a parameter or function result and the calling program must test this after every call. One cannot rely upon applications programmers to do this.

2. With subroutine libraries it is difficult to enforce standards. What tends to happen is that particular implementations become standard. In a competitive market this tends not to work. With statistical or mathematical libraries, the main users and providers of the software are academic institutions. They are not trying to make a profit from their software and have a strong incentive to standardise. Database software has been developed by and for commercial users, who have every incentive to put their own product on the market. They may be able and willing to produce software to a language standard but they are unlikely to accept another company's set of subroutines.

3. Subroutines provide very poor protection mechanisms. The user program may fetch a record into a buffer of a different format from that of the record, without this being detectable. This provides rich opportunities for corruption.

Subroutine libraries do provide the undoubted advantage of being able to use database software with an existing unmodified compiler. The next phase in the development of interfaces for programming languages attempted to retain this advantage whilst giving a higher level of protection and checking than subroutine calls. It involved the definition of language extensions that provided syntactic sugar plus an added level of protection to the basic subroutine interface. The extensions were handled by preprocessors.

Preprocessor Interfaces

Both the programming language Pascal and CODASYL database systems may be considered to have separate Data Definition and Data Manipulation Languages. In the latter case, they are explicitly separated, in the former they exist implicitly. All the syntactic constructions that can appear in the CONST, VAR and TYPE subsections of a Pascal program constitute Pascal Data Definition Language (DDL). The syntactic constructions that can occur within a block constitute its Data Manipulation Language (DML). The CODASYL database systems provide type definition facilities that are comparable in power to those of Pascal. It has three type construction operations in its Data Definition Language: cartesian composition of elements to form records, the formation of vectors out of repeating groups of elements, and the construction of sets of records. Its notion of a set is somewhat different from that of a Pascal set, but is more powerful and makes up for its lack of a reference type construction operator.

Where Pascal and CODASYL database systems differ is in algorithmic power and facilities for persistence. Pascal has no persistence facilities other than an untype-checked file input output . It is possible to declare files of any type in Pascal, but there is no check that a file is actually of that type when linked to a program. CODASYL allows all its datatypes other than cursors to be persistent. On the other hand Pascal has far greater algorithmic power. CODASYL DML has an impoverished control structure designed only to deal with error conditions, lacks expressions and is mainly made up of navigation, selection and assignment operations. The reason for this is that CODASYL DML is intended to be embedded in an existing algorithmic language, e.g. Cobol or Fortran. The host language is intended to supply the expression facilities and control structures necessary for the implementation of algorithms. The CODASYL DML provides the selection and assignment operations on the persistent data types provided by the DDL. Any calculations on or transformations of the data must be carried out by the host language on transient data.

The embedding of the DML in the host language is done using a preprocessor phase in which the DML is translated into a set of subroutine calls on the database system. The actual transfer of data between the database system and the program is effected by inserting declarations for buffers of the various CODASYL record types into the code of the source code of the application program. These are then passed as parameters to the DBMS in subroutine calls. The preprocessor can also automatically generate code sequences to check for error conditions after each call on the DBMS that will cause a branch to a user provided error handler on encountering an error condition. Having been preprocessed the program is passed to the COBOL or Fortran compiler and compiled and linked like any other.

The problem with this approach, which has been repeated with other database systems and other languages, is that it fails to overcome the dichotomy between temporary and persistent storage of data. It does give the programmer access to data of both types and allows both of them to be structured in a reasonable way. But they are still different. The temporary and persistent data belong to different types with different operations available on them. Persistent data has to be explicitly FETCHED before it can be operated on in an expression.

It may be said that this is partly due to the fact that DML is embedded in old languages like Fortran and more particularly Cobol, and that the operations provided on persistent data must therefore conform to the Cobol "style". This does explain part of the difficulty. Cobol's own data structuring facilities are less sophisticated than CODASYL DDL even though the latter is closely modeled on Cobol. There was probably little option but to make the record accessing facilities analogous to Cobol file i/o in that they require explicit fetches. In a language without reference types and dereferencing coercions there is a limit to how concisely one can express what is essentially a dereferencing operation. The end result, however, is to reproduce in a new form the old antithesis between inside and outside. Cobol was developed at a time when data processing was done on serial files on magnetic tape, the anatomy of CODASYL DML shows its ancestry.

Chapter 2

Algorithmic languages with database constructs.

Recently several attempts have been made to develop new algorithmic languages which incorporate database constructs. Examples are Pascal R, Plain, Aldat, TAXIS, DIAL and ADAPLEX [13], [14], [15], [16], [17], [18], [19]. In each case, the algoritmic portion of the language is closely based upon some existing algorithmic language, which has its syntax extended to support new data types deriving from some database model. The objective has been to combine the expressive and algorithmic power of the programming language with the data handling facilities of the database model. The database model has two conceptually distinct things to add to the language: data persistence, and a new set of type constructors and operators to support its view of data. In the case of the relational model the new type is the relation and the operations of the relational algebra the operators. Most algorithmic languages have the power to implement database models using existing data type constructors and algorithmic constructs. Relational database systems are, after all, written in conventional programming languages. The justification for making such facilities part of the language definition is that the primitive relational operations are sufficiently widely applicable and the cost of reimplementing them sufficiently high, that the added convenience of having them as built in constructs, compensates for the additional complexity it introduces into the compiler and run time support system.

In assessing these extended languages, one must judge them on three criteria:

1. Whether the new data model has been introduced in a way that conforms with the spirit of the original algorithmic language. If, for instance, the data model requires a new type constructor, can this be used in the same way as existing type constructors?

   Can it be composed with existing type constructors in type construction expressions?

   Can it be used in the same way in the declaration of identifiers?

   If the data model requires new operations are these dealt with in a way appropriate to the host language?

   Let us take the example of inserting a tuple into a relation. In the case of a business data processing language like BASIC or COBOL it would be appropriate to extend the language with a new type of statement to carry out the insertion, since these are statement based languages. One might end up with a statement like:

   INSERT tuple INTO relation

   In the case of a Pascal derivative it would be more appropriate to use an assignment statement with the relation on the left hand side and a concatenation expression on the right. One would end up with

a statement like:

relation := relation + tuple

In the case of an expression oriented language like C or Algol68 which already has a rich set of modifying operators one might overload one of these to do the insertion. In this case one would end up with a construction like:

relation +:= tuple

These points of style may seem trivial, but when a particular syntactic style is stuck to consistently it should bring benefits. The fewer the number of syntactic rules in a language and the greater their orthogonality, then the easier it should be to learn. One of the justifications for extending an existing language, other than saving one time on language design, is that there is already a body of experience in using the language. If any additions conform to its style or spirit then the additional effort required to learn about the new features is minimised. Finally, as Hoare has pointed out, programs in languages with simple rules are more likely to be correct [20].

2. Whether the rules about the persistence of data are applied in a consistent and orthogonal fashion. In a database oriented extension to an algorithmic language, one assumes that at least the new datatypes required for the database model will be persistent, but are these the only data types that can be persistent?

To produce a genuine integration of the database model into the programming language, persistence should be considered to be orthogonal to type. If one introduces relations into a language, one would not want all of these to be persistent. Some will be temporary relations used as a scratchpad for relational algorithms. Given that not all relations are persistent, orthogonality demands that not all persistent things should be relations.

3. Whether the extended language provides a clearly defined mechanism for binding database variables within a program to database objects persisting outside the program. It is worth reflecting for a moment on how programming languages have been interfaced to files. The binding between a programming language and a file is usually left until run time. The file is given an internal identifier, which is bound to an actual file either by the operating system command language, or by a call on an operating system subroutine that opens the file. This technique, banal as it seems, has the great advantage of enabling the same program to be run against many different files, and making it possible to delay the choice of which file to run against until after the program has started. In practice this flexibility turns out to be very valuable.

We can contrast this with the usual method of binding to external

subroutines. This is usually done in a separate linkage phase prior to execution. This approach works because the external routines constitute part of the algorithm of the program and as such are an invariant. The data that an algorithm works on is not invariant.

We will see that some database extensions to programming languages have made the mistake of treating persistent data as analogous to external subroutines rather than to files.

THESEUS

In the case of THESEUS the underlying language is EUCLID, itself a derivative of Pascal. The database model is the relational one. Shopiro argues in favour of this model that it allows flexibility in modelling the real world, without being biased towards any particular application. Without for the moment committing myself as to whether the relational model does have these advantages, I would suspect that the reason Shopiro and others have chosen it is its widespread popularity. That in its turn may well have something to do with its similarity to previous data processing techniques. The relational model is an abstraction of files or records, and as such is readily understandable to a data processing community which is used to crunching through files of records with COBOL programs. Indeed one can seek the origin of the model further back. The file of records, itself owes something in concept to a deck of punched cards. The operations of selection and projection, are not so different from what used to be done using electromechanical devices and punched cards prior to the use of stored program computers: natura non facit saltas.

THESEUS has two new data structuring facilities: A-sets and relations. A-set stands for Associative set. An A-set is a set of name value pairs, or a set of named values. Shopiro claims that they are a generalisation of the programming language notion of a record or the database notion of a tuple. Both of these contain named fields with associated values. The A-set differs from them in having a variable number of fields. In THESEUS all A-sets are created empty. They can later have name value associations inserted into them.

The names must be predeclared identifiers, similar to the value denotations in a Pascal enumerated type. They are not quite the same in that they are typed as shown in this example of a name declaration:

```
name partnumber:  integer ;
```

The type integer associated with partnumber acts as a constraint on the values with which the name can be associated. It would seem that this is necessary if you are to stick to Pascal type rules. Inserting name value pairs into A-sets is done as follows:

```
put partnumber  is 74151  in icval ;
```

This inserts the association between the name partnumber and the value 74151 into the A-set icval. There is also a remove operation to take a value out of a

set and a boolean valued function present which given a set and a name returns true if the name has a value associated with it in the set.

Shopiro asserts that the A-set is a more powerful concept than the tuple or record and marks a step towards artificial intelligence practice and that it brings databases closer to knowledge-bases. It is a step forward from the notion of a fixed format record, but only a small step. As implemented in THESEUS it has two definite weaknesses:

1. The names used in name value pairs have to be declared at compile time. This rules out the writing of programs which extend the range of associations in A-sets at run time. A knowledge base whose universe of discourse was known at compile time would be a rather mean thing.

2. The syntactic constructs used to implement insertion and deletion from A-sets are not orthogonal with that used for assignment which is a corresponding operation.

The other innovation (relative to EUCLID) in THESEUS is the introduction of relations as a data type. Relations are sets of A-sets in THESEUS. A-sets can be inserted into or deleted from relations, again by the use of special purpose non-orthogonal syntax. There is a facility for selection, which is termed restriction in THESEUS and a new iteration construct to allow the scanning of A-sets and relations. An interesting feature is that the programmer may define his own procedures to implement insertion and deletion. This allows checks to be made on insertion for attempts to insert illegal A-sets into the relation. There is a provision for a relation to be declared as external meaning that it will be bound to some external relation in a persistent database. This seems to be a particular weakness of the design. It results in programs that will only run against the external databases that are explicitly named in the program and it seems to rule out being able to link to databases dynamically on the basis of some run time condition.

PLAIN

PLAIN is another extension of Pascal. The extensions are in two directions, towards a language that is better adapted for writing interactive programs, and towards a language for data intensive programming. In addition it contains all the now usual extensions to provide modules, separate compilation and exception handling. The extensions for interactive programs are mainly in the area of string handling and input output. The most important feature here is a powerful pattern definition and matching capability. Patterns may be used to split up or combine strings, and to direct input and output.

The database extension is the provision of a type relation and a set of operators that manipulate it. The way that these are provided is syntactically much cleaner than in THESEUS and has a lot in common with Pascal R. These operators are as follows:

PLAIN DATABASE OPERATORS

Operator	Operation	Operand types	Result type
where	selection	relation, predicate	relation
=>	projection	relation, subset	relation
join	join	relation attribute	relation
*	intersection	relation	database
+	union	relation	database
-	difference	relation	database

Using these operators it is possible to construct database expressions whose whose result can be assigned to a database variable or act as an operand for another expression. Much of the expressive power of Algol-like language derives from the recursive nature of their expression syntax and it is important that any new types introduced into such languages should also be usable in a recursive expression syntax. PLAIN is much more successful in this respect than THESEUS.

Pascal R

Pascal R was the first extension of Pascal to include relational data types. Although the first, it is probably the most elegant. The integration of the relational data types with the existing types of the language is particularly well handled. Schmidt starts off well by basing his relations on an extension of an existing construct: the record. Pascal R relations are relations of records. Relations may have certain fields of the record designated as keys. The effect of this is to specify that the key fields must be unique. Only one record with the specified key must exist in the relation.

The language extends the Pascal assignment operator in a consistent way. The existing assignment operator

```
:=
```

is retained for assigning one relation to another. A new operator

```
rel1 :+ rel2
```

is introduced for inserting the contents of rel2 into rel1. Another operator

```
rel1 :- rel2
```

deletes all the tuples in rel2 that exist in rel1 from rel1. Replacement of tuples in a relation is effected by the

```
rel1 :& rel2
```

operator which replaces the non-key fields of those tuples in rel1 whose keys occur in rel2 by the coransponding fields from the tuples in rel2. A coercion operator is provided. The effect of

```
[ <record> ]
```

is to coerce the record into a relation of cardinality one.

Iteration over relations is handled by an extension of the Pascal for statement into the

```
foreach rec in rel do
```

where the rec is an iteration variable of the record type of the relation. The boolean constructions of the language have been extended with a couple of extra predicates

```
some rec in rel (<logical expression>)
all rec in rel (<logical expression>)
```

which mean that for some or all of the records in the relation the logical expression yields true. Finally the language provides facilities for expressions of type relation, the results of which may be assigned to another relation. For instance

```
[ each rec in rel : <logical expression> ]
```

yields the relation composed of all the records in the relation for which the logical expression is true.

As an addition of a new type constructor to Pascal, Schmidt's proposal is still one of the best thought out so far. Its weakness as with the other similar languages above, stems from a failure to identify the autonomy of two factors: the addition of a new data type relation and the addition of persistence. Persistence is sneaked in the back door as an attribute of relations, whereas in principle there is no reason why relations need to be persistent. The relational data type is a useful addition to the language quite regardless of whether relations are made to persist. A consequence of persistence coming like a thief in the night is that a lot of data is lost: the only things kept are the relations. Secondly, not enough thought is given to how we are to bind programs to chunks of persistent data. This is not specified by Schmidt.

ADAPLEX

ADAPLEX is a language developed at the Computer Corporation of America. It is intended to be a general purpose programming language with special extensions for the programming of database applications. It is the result of embedding the functional data language Daplex [21] in the advanced algorithmic language Ada. In their use of Ada as a base language Smith, Fox and Landers are in advance of other workers who have used Pascal. This is not just an advance in terms of fashion or modernity for its own sake. The modular structure of Ada lends itself well to the addition of extended facilities, its separate compilation facilities can be used to advantage in arranging the binding of databases to programs that operate on them. These sort of facilities had to be invented for PLAIN, in Ada they are already there. It can be argued also that the data model they have chosen is more advanced.

The functional data model was proposed by Buneman [22] and further developed by Shipman. It is now being used for a number of projects at the Computer Corporation of America. In [23] it is argued that the functional model is sufficiently powerful itself to provide a model for relational and network databases. It has the great advantage of being parsimonious in its conceptual structure. A single concept, the function or mapping is used to provide all data structuring and access operations. Given that Ada supports strongly typed functions, a purely functional extension to the language is very attractive.

ADAPLEX adds two new type constructors to Ada. The Entity constructor is similar to the record constructor that already exists in Ada. It builds a cartesian of named components. Unlike the Record type, which does the same the components of the cartesian are treated as functions from instances of the entity type to an instance or set of instances of their range type. If we consider the conventional record type in Ada or Pascal, it is evident that we can consider the syntax for field selection to be a means of constructing updatable functions from instances of the record type to values of the field type. What Daplex and after it ADAPLEX, does is to make the functional or mapping nature of this operation explicit in the syntax of field selection. Although the basic intention in Daplex is admirable, the same cannot be said for the syntactic implementation of the concept in ADAPLEX.

An example of an entity declaration in ADAPLEX is:

```
type person  is  entity
     name    : STRING (1..30);
     age     : INTEGER;
     phone   :  set  of STRING (1..11);
end  entity;
```

This syntax is clearly based upon that for Ada Record declarations. The syntax for accessing an attribute of an entity is however based upon that of Ada functions. So that the name field of a person entity John would be represented as:

```
name (John);
```

rather than as:

```
John.name;
```

There is a clear dislocation here between the two syntactic conventions which is likely to prove confusing. The other new type constructor is set. ADAPLEX allows the construction of sets of entities, strings or scalar types. The class of types from which sets can be constructed is the same as the class of types over which attribute functions of entities can range. The non-orthogonality of the set constructor is to be deplored.

In ADAPLEX persistence is associated with type. Values of entity or set types persist. It follows from what we said earlier that persistence is not a property that may be orthogonally composed with the other type constructors in Ada. Instances of record or array types may not persist as these may not be members of sets or ranges of entity attributes. On this basis one must say that the database facilities of ADAPLEX are not fully integrated with the existing Ada facilities.

In ADAPLEX, the binding between persistent data and a program occurs through the use of a new construct, the Module. A Module is an extension of an Ada package, with the exception that:

1. A Module, and the data in it may be shared by more than one program.

2. A Module is elaborated only once, at the occasion when it is used for the first time in the execution of a main program.

3. It is implied, though not clearly stated, that Modules persist and hold their data after the execution of a program.

The ADAPLEX manual is silent on whether a given program may run against several different Modules all of the same type. It seems that the Modules with which it is to be run are determined at compile time, or at very least during linkage.

Other Languages

The TAXIS language proposed by Myolopolous and Bernstein is presented as an extension to Pascal type languages. It seems that the system is in an early stage of development. As far as can be determined from the published material and conversations with the authors there is as yet no implementation of it. The design is still incomplete. Although an interesting semantic model, based upon semantic nets and abstract type facilities, is proposed, the syntactic and semantic details of how this will interface to an algorithmic language is glossed over. It is thus too early to give a definite judgement upon the particular points which I am interested in: the consistency of the interface with the existing language, the orthogonality of data persistence and the mechanisms for binding programs to databases. There is little in the published material on the language to indicate that the authors intended to implement persistence in an orthogonal way, it looks

as though they intend to limit it to the new data types that they will add to the existing language.

A couple of old languages whose development has nothing to do with databases, actually implement data persistence in an orthogonal if primitive fashion. Both APL and LISP implement persistence. APL has the concept of workspaces. A workspace is intended to be like a notebook, it contains the definitions of functions and variables along with the values of the latter. By use of the SAVE command the entire context that a user has been working in can be saved as a workspace in a library. By use of the LOAD command, a workspace may be loaded from a library to become the current workspace. Users have their own private library to which they have read write access, they may have read access to other libraries. The system also provides facilities to catalogue functions etc.

LISP provides two facilities for making programs and data persist. A user's workspace can be printed to a file, or saved as a core image. The printing to a file makes use of the fact that all LISP data has a standard external representation.

The popularity of both these languages owes much to their having been implemented as interactive systems. The provision for data persistence seems to have grown incidentally out of providing an interactive system. The systems have obvious shortcomings for database purposes. The workspaces are small. The facilities for sharing of data between users is poor. One cannot run a program against several different databases. If these faults could be overcome, this sort of interactive system provides a target for developers of persistence to aim at.

Chapter 3

Persistence as an orthogonal property.

In the previous chapter, a number of attempts to incorporate database constructs into algorithmic languages were criticised because they failed to identify persistence as an orthogonal property. Instead it was treated as an attribute of particular data types which were imported from the domain of databases into programing languages. The consequence of this is that algorithms which are to use persistent data must operate on different data types from those operating on transitory data. There is a very large body of algorithmic techniques which have been developed to use the data types and type constructors available for transitory data in present day algorithmic languages. It would obviously be rather nice if these could be applied to persistent data without either having to perform input/output operations or having to translate or contort the algorithms to conform to the relational model.

A fundamental design aim of the Data Curator Project was that programmers should be able to write programs in a way which is independent of the persistence of the data on which they operate.

Persistence of data is a continuously variable property from the transitory existence of results in the evaluation of an expression, via local variables of successive levels of procedure application, or associated data on a heap, to data that is retained on backing store for periods exceeding the execution of one program and ultimately to data whose usefulness exceeds the working lifetime of various programs which operate upon it. There is no reason why some particular point on this spectrum of data should exist, such that, on one side of this point the programmer must identify and operate on data in one way, and on the other side a different method or notation must be used.

Given this overall philosophy, that data has varying persistence, it is necessary for the owner of the data to indicate how long data should remain in existence. How may she do this? A number of mechanisms were considered:

i) Associate persistence with variables. This is unsatisfactory as local variables, particularly parameters of procedures must be capable of holding data of varying persistence if we are to achieve the goal of programmers being able to write algorithms which operate on data of any persistence. If we modify this idea slightly, to associate data with variables of global scope, it has more promise. It is dealt with in more detail under iii).

ii) Associate persistence with type. Clearly data cannot outlive the type which describes it. However this is only a bound, and not a sufficient description as within that bound instances of the same type may have widely different persistence. For example the type integer is

permanent in most languages. It would be distinctly embarrassing if all integers existed indefinitely, and explicit deletion would be cumbersome.

iii) Identify persistence by the generator used. [24] In this case a new generator, say db is introduced, and items generated with the db generator persist. This makes persistence two valued, or requires explicit deletion. Unfortunately it is not always known that an object is worthy of persistence at the time it is created. For example during the interactive design of some complex object, many temporary or experimental changes may be encountered, which will have similarly temporary data, but one of them, if the design is to progress, will become permanent.

A fundamental problem arises from this approach. Suppose a number of programs are to run on different occasions to operate on data identified as persistent. How do they reach these data? That is, how is it bound to variables? One approach is to arrange that variables in the outermost block (stack frame) are the same for all these programs or all the executions of the same program, and that, when the program is started, it has the value of this stack frame (execution record) loaded in the state it was in at the termination of the previous run. This leaves the problem of how to indicate to the program whether this is a "first" run, and how to test or designate that condition within the program. Further complexity arises if two projects wish to use some common data (e.g. a standard parts file) and separate instances of the same type of structure corresponding to the separate developments of each project. Ad hoc and cumbersome methods to solve these problems lead to unacceptable complexity in the language.

iv) Identify persistence by accessibility.

The mechanism by which we have chosen to support data persistence is an extension of that already used in programming languages with a heap. In these, items on the heap persist so long as they are accessible. An item is accessible if either it is referred to by an identifier that is in scope, or it is referred to by an accessible item. In order to support persistence it is necessary to:

i) Provide a set of identifiers which persist between program invocations and which can act as the roots from which persistent

accessibility is determined.

ii) Provide a means by which these can be bound to and accessed within different programs or program modules.

iii) Provide a means by which these identifiers may be initialised, and by which they may have new values assigned to them.

In the course of our research we have investigated a number of different mechanisms for setting up these persistent identifiers. The rest of this chapter presents, more or less in order of investigation, the alternatives that have been looked at.

Other Aims.

It was considered that a persistent language should provide a combination of the desirable features of database systems and programming languages. We considered the desirable features of database systems to be:

i) Precise description of data,
ii) Ability to construct models of real world systems,
iii) Identification mechanisms for data which are appropriate to large volumes of data,
iv) Data integrity - reliable long term storage,
v) Enforcement of constraints on the use of data - privacy controls and consistency constraints,
vi) Provision of views of the data to allow different users appropriate views,
vii) Provision for independent change of parts of the data definition from the programs that use the data,
viii) Provision for many programs to access the data,
ix) Provision for concurrent access,
x) Provision for identifying transactions which either achieve all their alterations to the data, or the data is left unchanged,
xi) Provision of general purpose query languages to permit convenient specification of simple requests for data.

From programming languages we have:

i) strict enforcement of type rules,
ii) precise specification of types,
iii) Uniform and consistent treatment of all data objects,
iv) Procedural abstraction of operations,
v) Provision of persistent libraries utilised by many programs,
vi) Association of data specification and functions on such data into classes,
vii) A small yet comprehensive set of basic types and operations on them.

As a further general design aim we required that the core of the language should

be small so that it was easily implemented and learnt. Use of an applicative language as a basis for the final language would have been too great a specialisation, particularly as it is difficult to see how a strictly applicative language program can update a structure so that some other concurrently operating program can respond to the change. The basis for the final language had therefore to be an algorithmic procedural language with as many of the required properties as could be found.

Choice of a language to extend

The language which most heavily influenced the early design work was Algol 68. There were a number of reasons for this. One was personal familiarity with the language. It was the only language of which we had experience that provided a kernel of features suitable for the addition of persistence. That it was the only one we were familiar with was not accidental. It was the only language readily available in this country that provided the sort of base that we were looking for. It provided, (at least in the Algol 68R implementation):

a heap with a garbage collector
strong type checking
modules with abstract types (segments)
concurrency provision

Other possible languages were available but had fewer facilities, or had the facilities but were not available. Although, as will be explained later, Algol 68 turned out to be an unsuitable language on which to implement a prototype persistent language, most of our initial ideas were formulated in an Algol 68 context.

Persistent Segments

Algol 68R provides a construct, the segment, which is very similar to the Ada module. A segment is a sequence of clauses and as such may contain declarations of indicants e.g., modes, procs, operators, variables. There is provision for the control of visibility through the use of import and export lists. A segment is defined as being WITH other segments in which case all of the exported indicants of those segments are visible within the segment being declared. At the end of the segment is an export list of indicants to be made visible to other segments. In adding persistence to Algol 68 the obvious starting point seemed to be the segment.

The first idea investigated was to extend the idea of segments so that segments might be thought of as programs, processes or collections of data [25]
They would be like programs in that they can be run.
They would be like processes in that they can be considered as being 'still there' after having run, waiting in a suspended state. Every segment could be considered to have associated with it a flag and a semaphore called FINISHED (of home segment) and DELETED (of home segment).

When the segment's execution reaches its outermost END it would set FINISHED to TRUE and execute a WAIT on DELETED. The effect of this would be is that all variables, procedures and mode declarations from the outermost block level of the segment remain in existence until some external segment SIGNALS DELETED.

Segments would be linked to one another via WITH and EXPORT lists, the syntax for which is given below.

SEGMENT

title{WITH title-, etc FROM albname}(sec){EXPORT name-, etc}FINISHED

Each title in the WITH list designates a segment in the album designated by albname whose exported variables, modes, and procs are considered to be visible within the segment being declared.

Each name in the EXPORT list designates a mode, proc, or identifier that is to be visible to other segments.

No segment can execute until all segments in its WITH list have already done. If the WITH list refers to segments in the home album then a cascade execution may be initiated.

These syntax and semantics are very similar indeed to those in Algol-68R the difference being that the sements were thought of as processes rather than components of a sequentially executing program. They were only vestigial processes, the notion of process just serving as a conceptual means of defining persistence. The result was that persistence was defined using concepts already present in the language (it contains the notion of process and semaphore already).

This approach had the great advantage of providing a tight binding between program and database. The data would be of modes defined in the segment, it would be operated upon by procedures and operators defined over those modes. The segment would provide us with both data abstraction and persistence. The persistent data would be the values of the variables in the segment and of everything on the heap reachable from these. Even at this stage in the investigation it was realised that there would be a necessity for the construction of multiple databases with the same modes and operators available in each of them. In Algol-68R a segment defines a set of indicants which are present in each program that uses the segment. Once you introduce the idea of persistent segments it becomes necessary to have the idea of segment instances. The segment is no longer defined just in terms of its indicants, it must also take into account the values of the variables in the segment. This introduces the need for the concept of a segment invocation [26]

A segment definition would be created by storing the source text of the segment in the curator. This definition might subsequently be compiled. Neither the definition nor its compilation would create any storage space or cause any execution to occur.

A segment would be run by invoking it. Invocation would be effected by the command:

INVOKE title1 AS title2 USING(title3=title4 ——, etc)

where:

title1	would be the name of a segment definition
title2	would be the name of the invocation being created
title3	would be the name of a segment definition in the WITH list of title1
title4	would be the name of a previous invocation of title3

The invocation of a definition causes two events:

1. the segment would be executed
2. all of its identifiers are persistently stored in an activation record

If any of the title4's have not yet been invoked the invocation will fail.

The executable code of the segment may access any of the exported indicants in the invocations that it uses in its with list. This implies that it will also be able indirectly to change variables accesses by exported PROCs, but which are themselves invisible.
An invocation cannot occur within a segment, it is a user level command.

With the idea of invocations, the concept of the segment was moving towards the idea of a Simula class. The concept of a segment served three functions:

1. Unit of separate compilation.
2. Method of visibility control.
3. Unit of persistence.

With the existence of invocations, a segment definition became similar to a type or mode declaration. It defined the fields of a class of segment instances. At this point it became clear that there would be an element of redundancy in the syntax of a persistent dialect of Algol 68 built on these principles. In many ways the segment was beginning to look rather like a structure mode. Both contained named fields, both could have instances on the heap. In the interests of parsimony it seemed sensible to merge the two concepts [27]

In order to get this effect one had only to drop the STRUCT concept from the

language and allow exported indicants of segments to be refered to in the same way as structure fields (with the OF construct). At this point it was felt that one might as well call these new constructs CLASSES and be done with it. The logic of persistent modules leads to Simula like constructs.

Albums

Classes were to reside in albums. An album was to be a persistent heap with a directory structure in it. The entries in this were to be class definitions and class instances. Anything reachable from a class invocation or instance, would persist.

The model we had arrived at was to have databases structured into albums. These would contain named class instances and class definitions. The class definitions would provide our unit of compilation and visibility control, the class instances on the heap would provide our basic data structuring tool.

Convergent Evolution

Our arrival at this model does not seem accidental. Other attempts to deal with persistence in an orthogonal fashion seem to have led researchers to similar conclusions as to the basic architecture to be employed.

The Smalltalk [28] system, developed at XEROX Palo Alto Research Center originally for computer aided learning experiments has a number of similar features. Smalltalk is based upon a message passing paradigm. Objects pass messages between one another in order to effect computation. At first sight this seems to make the language very different from the procedural languages we are more familiar with. On closer examination one sees that message passing just substitutes for procedure calling; it is more powerful, but it lies along the same axis of the language as is normally occupied by procedures. The replacement of procedures by 'methods' invoked by messages is orthogonal to other aspects of the language design. The interesting thing for us about Smalltalk is one of its least remarked features - it implements data persistence in an orthogonal fashion.

It is an object based system, the objects being instances of classes. Objects persist if they are refered to from one of the roots of the system. These roots are the entries in a system dictionary of class names and variables. The variables have as values references to objects. The class acts as a means of defining abstract data types. Its fields can only be accessed indirectly via methods. The main feature that Smalltalk classes have that ours did not was the concept of a superclass whose attributes could be inherited by its sub classes. Some of the same functions could be achieved by our import of segment / class instances. Both mechanisms allow part of an environment to be made accessible within another environment. The Smalltalk mechanism has the added advantage of providing something analogous to the database concept of a view [29] in that superclasses can be considered to be views of subclasses. Despite our model being somewhat less powerful, its general structure was very similar to that adopted in Smalltalk.

Smalltalk was developed as a novel interactive programming environment, it was

not specifically intended for data intensive applications. The total address space provided was small, 64k of objects [30], though this may be extended in later implementations. It is envisaged that extensions of algorithmic languages to provide persistent data should allow at least 32 bit object addressing.

ELLE

As far as we can determine, the only system mentioned in the literature, other than our own, to specifically address the problem of providing data persistence as an orthogonal feature is ELLE developed by Orsini and his co-workers in Pisa [31] The authors of this system point out the inhomogeneity of treatment of temporary data in conventional programming languages. They acknowledge that there have been attempts to add persistence to programming languages, but allege (correctly in my opinion) that most of these do no more than incorporate an existing relational Database Management System into a programming language. In contrast to this, they

> believe that there is another approach which is worth investigating. That is, to start with a programming language and design a programming system where it is possible to use persistent complex data structures without resorting to an expensive tool such as a DBMS. The system should provide an interactive environment and a uniform use of all data, irrespective of its persistence or temporariness.

The language that they chose to base ELLE on is ML [32] It is a strongly typed, higher order, expression language. It provides for references as a means for sharing data. It incorporates the idea of environments

> both to control the interactions among different applications using common data, and to deal uniformly with persistence without resorting to specific data types.

The basic data structuring tools provided are labelled tuples, which are analogous to record or structure types, lists, and discriminated unions. A powerful abstract type facility is present which allows types to be constructed in terms of a base type, such that the new type is isomorphic to the base type and inherits its operations. The language allows overloading of operators over types.

The language ELLE provides persistence by means of environments. An environment is similar to the idea of a block in Algols except that it is treated as an object. It is a set of bindings between names and values where the values can be other environments. Environments can be refined by starting with an environment and generating others by adding new definitions. Because they are objects, environments can be nested. Environments persist so long as they remain bound to a name in an outer environment. At the outermost level, they are all nested within a global environment which is the root from which all persistence is derived.

It will be seen when I come to deal with our specification of NEPAL that our notion of what a persistent language should be like and that of the Pisa group are strikingly similar. Such convergent evolution can either be taken as

confirmation that our approach is basically sound, or that given the technical culture that at present exists in computing only certain paths of evolution are open to us be they good or bad. Whether the optimistic or pessimistic view of this convergence is justified only time will tell.

Chapter 4

Forays into Persistence Architecture

If the progress of development of database and programming language research had been a little faster, so that this research was started half a dozen years earlier, it would have been assumed that a persistent language system should be designed to run on a timesharing mainframe. But the spirit of the age moves on, even if mundane reality lags. By the time this work started (October 1979) it had entered the world of distributed systems, whence we attempted to follow it. Material constraints eventually remainded us that as Althusser said history permits no essential sections [33]. We designed the system to work over a local area network, but since we did not have a local area network we were forced to operate it on a single timesharing system.

Our basic assumption was that programs in persistent languages would be executed on single user personal computers. The persistent data would reside on a database server on the local area network. We called this machine the Data Curator. We assumed, as it turns out, without much foresight, that the personal machines (henceforth client machines) would be short of memory and that the portion of the run time support for the persistent languages residing on these machines would have to be small and simple. The Data Curator would be a 32 bit machine with more memory which would hold the more complex parts of the run time support.

Our initial model for the execution of a persistent program relied upon the idea that in all but the most primitive computers there is some form of memory protection. Some addresses are legal within a given context and others are illegal. This was seen as a possible way of treating data on the heap in a uniform way independent of its persistence. A mapping would be set up between the set of illegal addresses on the client machine and a subset of the persistent data held on the Data Curator. When a client machine logged in to the data curator it would specify the context or environment in which it intended to run in terms of an album and a segment or class instance within that album [34]

Along with information about the desired context, the client would pass information about its addressing structure the range of its legal and illegal addresses etc. In response the curator would send to the client a seed illegal address. This would be treated by the client as a reference to the base of a data structure from which all other legally accessible heap items were to be reached. If this data structure were a class instance the next action would be to call one of its procedural fields. This call would fail because it was directed at an illegal address. The address error would be trapped by the run time support software in the client machine which would inform the data curator that it had had an address fault at such and such an illegal address. The data curator would respond to this by sending to the client a persistent data item associated with that illegal address. Before sending it the Curator would overwrite all fields within the object that were references to persistent items in the database with illegal client addresses. The client would place this object on its heap and retry the instruction. By this means persistent objects held in the Data Curator would be brought into the legal address space as and when they were required.

A key concept in this process is that of a Persistent Identifier (PID). Each persistent object, (string, record, array) kept by the Data Curator would have a PID associated with it. In a later section the requirements for persistent address spaces will be examined, for the moment we will just consider PIDs to be persistent addresses to be used for update or recovery of persistent data objects. In order for the address faulting process described above to be used as the basis for an object oriented virtual memory, the data curator would have to maintain a data structure that mapped from PIDs to client machine local addresses. This Persistent Identifier to Local Address Map (PIDLAM) could be considered a relation with three columns.

Whenever an object was sent over to the client the following algorithm would have to be executed:

```
for each pid in object do
  if pid in PIDLAM then
    if legal.address = nil then
     field := illegal address
    else
     field := legal address
    fi
  else
   insert (pid, nil, next illegal) into PIDLAM
  fi
```

Whenever a client signalled an address fault, the curator would have to execute :

```
proc fault (illegal addr)
  search PIDLAM for (illegal addr)
   if legal address = nil then
    send over (object)
   else
    inform client of (legal address)
   fi
```

In order to carry out these functions the Data Curator was obviously going to need detailed information about the structure of the objects that were being sent over to the clients. This seemed to imply that it was going to have to have knowledge that would normally only be available to a compiler, which in turn implied that when programs were being compiled the compilers running in the client machine would have to enter into a dialogue with the Curator. They would have to inform the Curator about the types of the objects declared in segments under compilation, they would have to ask the curator about the types and procedures declared in segments that were imported into the segment . Once the segment had been compiled, the code and any constants associated with it (and perhaps also the source) would have to be stored in the curator. In short

the Curator would have to act as a database resource to compilers as well as application programs. At this point a complication was introduced. It was considered that persistence could be added to any Algol-like language which had a heap.

Multi Language Support

Ideally one might wish to design a new language with the requirements of persistent programming specifically in mind, but new languages have disadvantages.
It costs time and effort to learn a new programming language. It takes time and effort to build a compiler for a new language. A new language lacks the base of ready written software that is available for established languages. It was therefore desirable to make the minimum change possible to existing languages to make them persistent. A few extra constructs might be necessary to handle an outer scope required for persistent objects, but the basic syntax of the language with persistence should be upwards compatible with that of the language without persistence.

Given the model of the Data Curator as a server on a Local Area Network, it seemed desirable that it should be able to support programs in more than one language. A user should be able to write programs in persistent versions of for instance, Pascal or Algol68, and use the Curator to store the data associated with these programs. From there it was easy to make the bold leap of demanding that the data stored from a Pascal program should be usable in an Algol68 one and vice versa. This, after all, can be done if you store data in ordinary files.

The big difference of course is that files are untyped, whereas program heaps are typed. If one were to use Pascal data in an Algol68 program, then the Algol68 program would need to be able to view the Pascal data in terms of an Algol68 type structure. The Pascal types would have to be represented to it as Algol68 modes. Given that we had already recognised the need for a modular structure that allowed modules to import and export type definitions, it did not seem impossible that an Algol68 segment should be able to import types originally declared in a Pascal module. What was required was a canonical representation of types. The Algol68 and Pascal compilers would inform the data curator of the types they encountered using this canonical representation. When compiling a module in the context of other modules they would be sent information about these modules in canonical form. An attempt was therefore made to design such a canonical representation [35]

Cannonical Representation of Types

We wished to be able to construct types in as general a way as possible and have some means of specifying when types were equivalent and when they were not. We wanted the rules for type construction to be sufficiently general to be able to handle any particular type construction rules that we were likely to come across in the various super Algols that we were likely to support.

We assumed that we had a pregiven set of base types and wanted to be able to construct new types from these.

A plausible set of base types would be:

INTEGER
REAL
CHAR
BITS
VOID
ATOM
ALL*

(* only refs to all were to be allowed)

OPERATORS

We wish to have two relational operators on types: includes and matches . For all of the base types t

t includes VOID

and

ALL includes T for all types T

and for all T

T matches T

and

B matches C iff C matches B

A type is a set of values. The type construction operators are used to compose these sets out of other sets. In order to avoid self reference and Russell's paradox we must insist that no type may be a member of a type.

Type Construction

New types can be constructed out of base types by the application of the operations distinguishing, referencing, arraying, mapping, union, intersection, composition and ordering.

Naming

Types may be given names by the operator bind which associates a name with a type expression. Once a name has been bound to an expression then that name can stand for the expression in other expressions. It must be understood that the name becomes the name of a type it is not the name of a variable or constant of that type.

Let n bind T then n matches T

Distinguishing

In order to be able to construct enumeration types and to cater for the semantics of named types in Pascal we need a distinguishing operator **dist**. When this is applied to a type expression it yields a new type with the following properties:

1) there is an isomorphism between the values in the original type and the values in the type yielded by applying **dist** to it.

2) the intersection between the set of values in the original type and the set of values in the type yielded by applying **dist** is VOID .

Refing

There is a type construction operator **ref** such that:

Let REFA= **ref** A and Let REFB= **ref** B then

REFA **matches** REFB **iff** A **matches** B

Associated with the type constructor **ref** there is a metaoperation on the set of values belonging to the class of types constructed by **ref**, **dref** which yields a value of the type from which the reference type was constructed.

Union

Let T = Q **union** R and Let P = R **union** Q

Then :

(T **matches** P) and (T **includes** Q) and (T **includes** R)

Intersection

Let T = Q **intersects** R then

if P = R **intersects** Q then
(P **matches** T)and(Q **includes** T)and(R **includes** T)

Note

Union and intersection operate on sets of values. When T is formed from the union of Q and R the members of T are not Q and R themselves – but the union of the members of Q and R.

Difference

There is an operator minus that takes two sets of values and yields their difference.

Ordering

Languages such as Pascal provide for the construction of ordered types, other than the base types. The cannonical representation thus required some method of specifying that certain enumeration types were to be considered ordered.

```
Let T=Q    order P and

Let P=R    order Q  then

T  intersection P  matches VOID

and P  matches T is false

but   T  includes    Q

and T  includes   R

and the relation  q<r in T is true for all q in Q and all r
in R
```

By extensions of ordering and intersection, subrange types may be constructed. We need two new operators above and below:

```
Let A= T  above x

Let B= T  below y
```

Where x and y are values of type T and T is ordered

Then x is a greatest lower bound on the values in A and y is a least upper bound on the values in B. And we can construct the subrange x .. y as:

```
A  intersection B
```

Powersets

```
Let P =  power T
```

We can form a type whose members are the powerset of the

members of another set. These are equivalent to the sets in Pascal.

Composition

Structures can be composed by the operations of Cartesian composition and simple composition Simple composition is used for record construction in those languages in which the only way to access the fields of a record is by an expression or statement that includes the field name. Cartesian composition is used in languages such as Algol-68 in which assignment of a collateral clause to a structure is allowed. Here access to the fields is implicitly determined by field order.

Simple Composition

Let T=Q comp R then T matches R comp Q

Cartesian

Let T=QxR then T matches RxQ is false

Fieldnames

The fields of a Cartesian composition must and the fields of a simple composition may be named using the field operation. This gives names to fields of a composed type and must not be confused with the bind operator which names types.

Maps Functions and Arrays

Let M = I map O

This creates a mapping type M which maps values of type I to values of type O. Conceptually, arrays and functions are instances of mapping types.

If F= I fn O then M includes F and
If V= I array O then M includes V

A more detailed and formal specification of the semantics of the representation is given in [36]

Testing the Representation

The canonical representation was intended to be capable of representing any type that could be represented by the syntaxes of Pascal or Algol68. As an initial test of its power it was decided that an attempt would be made to hand translate sections of Pascal type declarations and Algol68 mode declarations into the the canonical representation. In order to ensure that the samples of Pascal and Algol68 should be realistic, sections of code from Pascal and Algol68 compilers were chosen [37] [38]. In both cases it proved possible to translate the source language into the canonical representation.

An Example of The Application of the Canonical Type Representation to Pascal.

Of the two test examples, the Pascal is the more readable, and is reproduced here to give some idea what the canonical represetnation would look like.

We took as an example a series of type declarations given in the book 'Structured System Programing' by Welsh and Mckeag [39] The types in question come from the type checking part of a Pascal compiler.

```
TYPE

   TYPENTRY = ↑TYPEREC ; IDENTRY = ↑IDREC ;

   TYPEFORM = (SCALARS,ARRAYS) ;

   TYPEREC = RECORD
               NEXT : TYPENTRY ;
               REPRESENTATION : GENERATE.TYPEREPRESENTATION ;
               CASE FORM : TYPEFORM OF
                  ARRAYS :
                     (INDEXMIN,INDEXMAX : INTEGER ;
                      ELEMENTTYPE : TYPENTRY)
             END;

   IDCLASS = (TYPES, CONSTS, VARS, PROCS) ;

   SETOFIDCLASS = SET OF IDCLASS;

   IDREC = RECORD
             NAME : ALFA ;
             LEFTLINK,RIGHTLINK : IDENTRY ;
             IDTYPE : TYPENTRY ;
             CASE CLASS : IDCLASS OF
                CONSTS : (CONSTVALUE : INTEGER) ;
                VARS   : (VARADRESS  : GENERATE.RUNTIMEADDRESS) ;
                PROCS  : (LINKAGE    : GENERATE.PROCLINKAGE) ;
           END;
```

Fig 4.1 Pascal Source

What follows is the canonical representation of the Pascal source given in Fig 4.1.

```
TYPENTRY  bind ( ref TYPEREC)

IDENTRY  bind ( ref  IDREC)

TYPEFORM  bind (SCALARS  order ARRAYS)

              SCALARS  bind ( dist ATOM )

              ARRAYS   bind ( dist ATOM )

TYPEREC  bind (
        (NEXT  field TYPENTRY) x
        (REPRESENTATION  field GENERATE.TYPEREPRESENTATION) x
        (
           (
              (FORM  field (ARRAYS  in TYPEFORM)) x
              (INDEXMIN  field INTEGER) x
              (INDEXMAX  field INTEGER) x
              (ELEMENTTYPE  field TYPENTRY)
           )
            union (FORM  field  (SCALARS  in TYPEFORM))
      )
    )

TYPES  bind (dist ATOM)

CONSTS  bind (dist ATOM)

VARS   bind ( dist ATOM)

PROCS  bind ( dist ATOM)

IDCLASS  bind (TYPES  order CONSTS  order VARS  order PROCS)

SETOFIDCLASS  bind
        ( powerset IDCLASS)
```

```
IDREC  bind (
          (NAME  field ALFA) x
          (LEFTLINK  field IDENTRY) x
          (RIGHTLINK  field IDENTRY) x
          (IDTYPE  field TYPENTRY) x
          (
             (
                (CLASS  field (CONSTS  in IDCLASS) x
                (CONSTVALUE  field INTEGER)
             )
              union
             (
                (CLASS  field (VARS     in IDCLASS)) x
                (VARADRESS  field GENERATE.RUNTIMEADDRESS)
             )
              union
             (
                (CLASS  field (PROCS  in IDCLASS)) x
                (LINKAGE  field GENERATE.PROCLINKAGE)
             )
              union
             (CLASS  field (TYPES  in IDCLASS))
          )
       )
```

Because this initial test seemed to show that it was possible to translate Pascal and Algol68 type declarations into our canonical representation, a detailed specification of the translation rules to be followed in mapping from Pascal to our representation was drawn up. On the basis of this specification a student hired over a summer vacation produced a translator that accepted Pascal type declarations as input, and generated an output stream of canonical representation of the same types. The output of this translator was taken as input to a process that stored the type information in a database. This process termed the Indicant Manager, was intended to act as a server to compilers and maintain a persistent compilation environment. Further details of the indicant manager are given in [40] [41] It was intended that the Indicant Manager would also be able to determine the equivalence of types. Given two type expressions in our canonical representation or type algebra, it would say whether they matched.

Slightly more formally:

Let A, P, C be the grammars of Algol68, Pascal and our canonical representation.

Let a, p, c denote strings produced by these grammars.

Given two functions:

```
TAC(a -> c)
TPC(p -> c)
```

Which translate between Algol68 and Pascal respectively and the canonical representation. And given two functions:

```
MA(a, a -> bool)
MP(p, p -> bool)
```

Which return true if a pair of type descriptions in Algol68 or Pascal respectively are equivalent according to the existing semantics of these languages, we wish to construct a function:

```
MC(c, c -> bool)
```

Such that:

$$MC(TAC(a_1),\ TAC(a_2)) \text{ iff } MA(a_1, a_2)$$

and

$$MC(TPC(p_1),\ TPC(p_2)) \text{ iff } MP(p_1, p_2)$$

Examination of the type matching rules of the two source languages showed that this would only be possible if the set of strings yielded by the application of TAC to Algol68 structure types and the set of strings yielded by application of TPC to Pascal record types were strictly disjoint, and that if

- s were an Algol68 structure type
- r was a Pascal record type

then

$$MC\ (TAC(s),\ TPC(r)) = \text{false for all s and r}$$

This is because in Pascal record types are equivalent if they have the same name irrespective of their structure, whereas in Algol68 the types are equivalent if their structures are equivalent irrespective of differences in names. One might expect that in Pascal all types of the same name would have the same structure, but a close reading of the Pascal Report[42] reveals that this is not the case. The following is not excluded by the Report:

```
type thing=0..9;
function nextthing( athing:thing):thing;
  type thing=10..19;
  var temp:thing;
  begin
      if athing=19 then temp:=10
      else temp:=athing+1;
       nexthing:=temp;
   end;
```

Our type algebra provided the means for constructing functions TPC and TAC that would abide by these rules, since we provided the operator dist that would construct types distinguished on the basis of their names. If this were used for the construction of Pascal types but not for Algol68 types the type semantics of the two source languages could be preserved. The question was, would it be useful to construct a canonical representation of types in two languages, which effectively prevented the exchange of the major sort of data structures produced by the two languages. It would be possible to match arrays and base types, and thus to exchange data of these types, but a great deal of the benefits of exchangeablity of data between languages would be lost. This realisation put a questionmark over the future of the model of data curator on which we were working.

Compiler Troubles

We finally decided that the model of a Data Curator maintaining language independent typed data was infeasible after examining available compilers for Algol68 and Pascal. As an exercise to gain familiarity with Algol68 compiler technology we implemented an Algol68C [43] compiler for Vax. This was based upon the existing Algol68C compiler front end developed at Cambridge University. The compiler produces an intermediate code termed Z-code.
This is a very low level abstract machine code, from which all type information has been deleted. Algol68C did not support a heap with a garbage collector. Because of the low level of Zcode and the lack of a garbage collector it was evident that the Algol68C compiler could not serve as a basis for a persistent implementation of Algol68. Two Pascal systems were also investigated. The UCSD Pascal compiler was examined and it was decided that this was too long and poorly commented to be easily modifiable. An interactive Pascal system termed COPAS, was obtained from Sheffield University and ported to the Vax. This was even longer and more difficult to modify than the UCSD compiler. It was concluded that the available compilers for Pascal and Algol68 were not readily modifiable to work in a persistent environment. Writing entire new compilers for these languages would have been impossible with the labour power assigned to the project. Since existing compilers of Algol68 and Pascal could not serve as a basis for the research, it seemed unlikely that there would be a requirement for a canonical representation of the types of these languages, so research into canonical representations was halted.

Keeping it Simple

Having thrown out canonical representation of types, it no longer seemed necessary for the data curator to know about the types of the data that it stored. The sole purpose for this had been to vet the exchange of data between different language systems. With this exchange ruled out, a language system should be able to keep track of the types of its own data. Secondly, in view of the very rapid fall in the costs of semiconductor memory, it no longer seemed to be necessary to postulate that client machines would have small memory spaces. Our original notion of a Data Curator was of a machine that would look after persistent data, data types and compiler environments, and also the run time environments of programs running on client machines by means of a PIDLAM. If memory was cheap it would make sense to move the PIDLAM out to the client machine, so that advantage could be taken of parallelism in execution of programs. This would enable the Data Curator to be reduced to a vestigial machine that merely acted as a record server for run time support systems running on client machines. In fact it seemed that the Data Curator could be reduced to the Chunk Management System to be described in a following chapter.

Chapter 5

PS-Algol: a Prototype Persistent Language

After it had been decided that the design of a Data Curator capable of supporting both Algol68 and Pascal was not a viable research project, and that there were no readily available Pascal or Algol68 compilers that could be readily modified to produce a persistent language implementation, it became necessary to search for some other language and compiler that could be so developed.

The locally developed languages IMP77 [44] and ML were rejected, the first because of its absence of a heap and poor type protection, the second because the compiler was not yet available. The language chosen for further development was S-Algol [45].

S-Algol is a language developed at St Andrews University, Scotland. It belongs to the Algol tradition, and has a very concise and orthogonal syntax. The guiding tenet of its design was that power is gained through simplicity and simplicity through generality. In expressive power and orthogonality it is somewhat above Pascal. Its most important features are its data structuring facilities.

The base types of the language are **int**, **bool**, **real**, **file**, **string** and **pntr**. Strong typing is ensured by a combination of compile time and run-time checks. Strings support the operations of concatenation and substring selection.

The type constructors are **vector**, and **structure**. vectors are updatable mappings from dynamic ranges over the integers to one of the base types, or to a vector type. Multidimensional arrays may be formed by composing the vector construction operations. Structure classes are ordered cartesians of named fields belonging to one of the base types, or to a vector type.

Pointers are access descriptors which can reference instances of any of the structure classes but which may not reference instances of base types or vectors. The language provides the operations of field subscription and run-time type verification on pointers. This arrangement makes it possible to write algorithms to manipulate pointers without the need to know the type of the referend.

The program fragment:

```
structure cons( pntr hd,tl)
structure string.atom( stringval)
structure nil
let a.string = " Dumpty"
let a.pointer = string.atom("Humpty" ++ a.string)
let another.pointer := cons(a.pointer, nil)
while another.pointer isnt nil do
 begin
        write if another.pointer(hd) is string.atom
              then another.pointer(hd,val)
              else "Not a string"
       write newline
      another.pointer := another.pointer (tl)
end
write "End of list"
```

would produce output:

```
Humpty Dumpty
End of list
```

Note the following:

1. Identifier declarations are introduced by the word 'let' and are initialising.

2. The type of an identifier is given by the type of its initialising value.

3. Constant identifiers are initialised using '=', variables using ':='.

4. The operators 'is' and 'isnt' are used to check the class currently referenced by a pntr.

5. The declaration of a structure class implicitly declares a generator function of the same name, whose application yields an instance of the class.

6. Subsription of arrays and field selection of structures is uniformly performed by

   <object> (<selector>)

   where <object> is a pntr or vector and the selector is a fieldname or index, or a list of fieldnames and indices. This syntactic form, which is the same as is used for function application serves to emphasise the sematic substitutability of subscription and function application within expressions.

The Heap

In S-Algol all compound data objects: strings, structures, and vectors are generated on the heap. There is a garbage collector which preserves objects reachable from identifiers currently in scope.

Implementations

S-Algol has currently been implemented on the following sytems.

Machine	Operating System
PDP11	Unix
VAX	VMS
Z80	CP/M
PE3220	Mouses

It was chosen as the starting language because:

1. It is a small simple language with a small simple compiler.

2. It provides a heap with proper garbage collection.

3. The generic pointer type along with run time type checking allows the construction of generic procedures, which were anticipated to be useful for database work.

4. The compiler writer was within reach.

PS-Algol: a bootstrap for Nepal

Starting from S-Algol an extended language termed New Edinburgh Persistent Algorithmic Language (Nepal) was designed [46] This involved significant changes to the scope rules and type construction mechanisms of the language. A detailed account of Nepal is given in a following chapter. The existing S-Algol compiler, written in S-Algol, was modified over the course of a couple of weeks to recognise the syntax and scope rules of Nepal, though not to generate code for it. Nepal presupposes the existence of persistent environments called groups, in the context of which new program modules are compiled. We call this compilation in a persistent context. It implies that the compiler's symbol tables and type information are stored in a database, which would obviously be easier if the compiler itself were written in a persistent language. In the long run it was intended that the Nepal compiler would be written in Nepal, the problem was how to bootstrap ourselves into the position where that would be possible. The course chosen was to design a second language PS-Algol (Persistent S-Algol) which differed in only a minimal respect from S-Algol, but which supported persistent data [47] The PS-Algol compiler would be written in S-Algol. The Nepal compiler would then be written in PS-Algol.

PS-Algol programs had, at least for the first versions of the compiler, to be compilable in a non-persistent context, because the PS-Algol compiler would be written in S-Algol which has no facilities for database accessing. The Nepal

Bootstraping Nepal

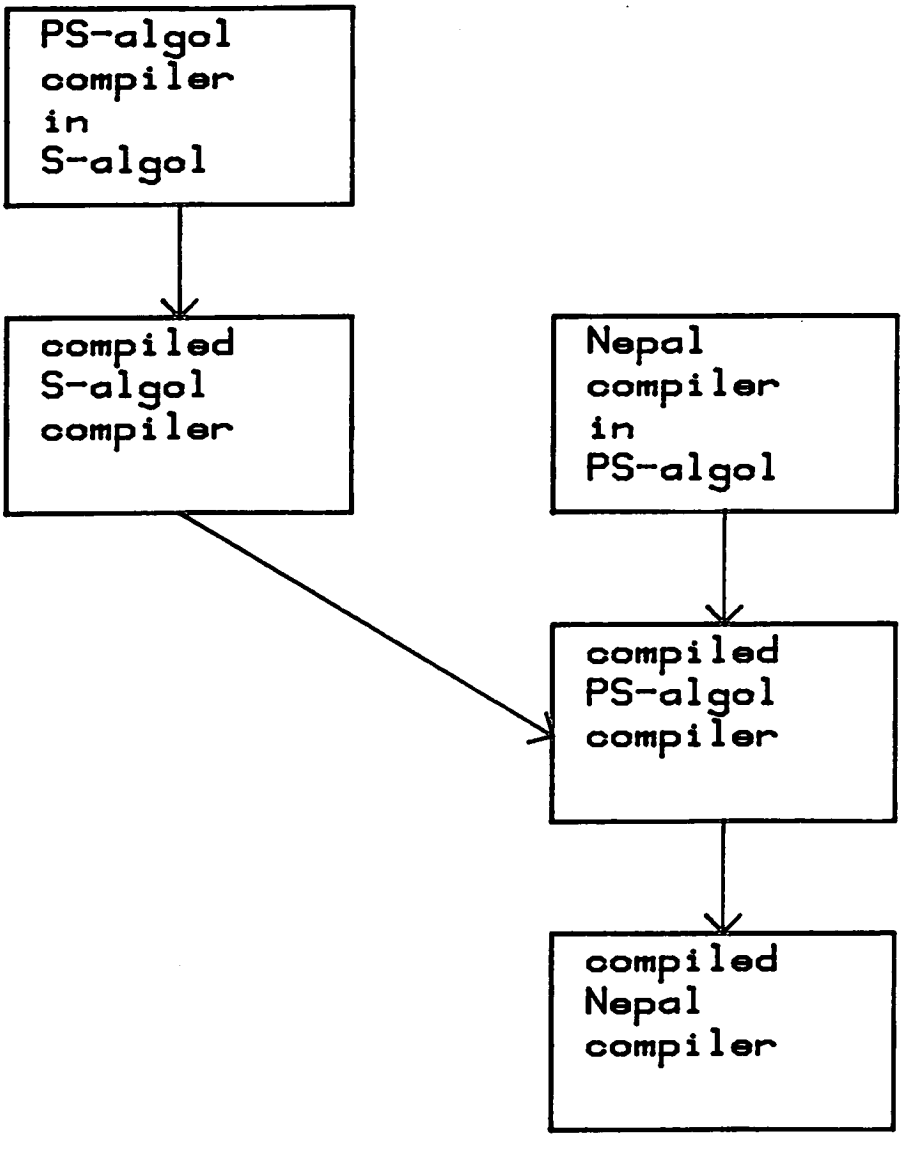

Fig 5.1

compiler written in PS-Algol would have the necessary facilities to compile code in a persistent context.

The design of PS-Algol was driven by two factors: the need to have enough power to provide a compilation environment for Nepal, and the need to be as close to S-Algol as possible in order to simplify implementation. We consider that the language resulting from these pressures has been pared down to just about the minimum set of additional features required to support persistence. We consider these minimum features to be:

1. A persistent heap.

2. A method of linking programs with preserved heaps.

3. A mechanism to delimit transactions.

An additional desirable facility is:

4. An associative store.

A persistent heap is one on which a datastructure built in one run of a program may be preserved to be used in other runs of the same or other programs. A method of identifying these preserved heaps and binding to them is then necessary. A transaction mechanism makes it possible to ensure the integrity of data. It ensures that all the changes made during a transaction are effected or that the data is restored to its original state .

Arrays indexed by scalar types provide an updatable mapping from a range of a discrete type to some other type. Associative store is a further generalisation of arrays. It allows the construction of updatable functions sparsely indexed by other types such as strings. The provision of associative store in a language does not imply that it is provided by hardware, just that the abstract machine presented to the language user by the language supports association. Although associative store is not necessary in a database language, since it can be emulated in software written within the database language itself, its effect is required often enough to make its inclusion desirable.

PS-Algol as an extension to S-Algol

PS-Algol is a derivative of the language S-Algol. Syntactically it is identical to S-Algol. The only visible extension to the language is the addition of a number of new predeclared procedures and a predeclared type. The effect of these, however, is to greatly increase the power and generality of the S-Algol heap. These procedures provide:

1. A method of associating a program heap with a named database.

2. A method of committing a transaction upon the database thus simultaneously updating the database and preserving a subset of data on the heap.

3. An associative store.

4. A universal nil pointer.

The procedures are as follows:

```
procedure open.database(string user.name, password, db.name->bool)
```

If the correct user password is supplied, this opens a database belonging to the specified user, with the db.name given. If the database does not yet exist it is created. If another user currently has a transaction open on the database then the open fails. Any failure is indicated by returning false otherwise a mapping is established between the database and the program's heap and true returned.

```
procedure root.table(->pntr)
```

This procedure returns a pointer to an instance of the predefined class 'table' which provides an associative store facility. The internal structure of this class is hidden from programmers using PS-Algol . However, the procedures that follow allow operations on instances of this class. Associated with each database there is a distinguished table that is returned by root.table.

```
procedure nil (-> pntr)
```

S-Algol has no predefined nil pointer, so one generally declares a local nil pointer and uses it to designate the end of lists etc. The weakness of this in a persistent environment is that there would no longer be a unique nil pointer, but separate ones for each program that ran against the database. This procedure returns a system wide nil pointer.

```
procedure enter(string key; pntr table, value)
```

This enters the parameter value into the table using the key. The value, being a pointer, can reference any arbitrarily complex data structure, including another table. The effect of entering a value/key pair is to set up an association between the key and the value, allowing the key to be used for later retrieval of the value. If the value is nil the effect is that of deleting the item from the table.

```
procedure lookup(string key; pntr table -> pntr)
```

This returns the value last associated with the key in the specified table. If no value has been associated with this key, nil is returned.

```
procedure table(->pntr)
```

This returns a pointer to an empty table.

```
procedure scan(pntr table,status; (string,pntr, pntr -> pntr)user)
```

This scans the table by applying the function user to every key in the table. This function takes as its parameters a key, a table and a pntr to a status record. It returns a new status record. The purpose of the status record is to achieve the same effect as an own variable in Algol 60. The iteration ceases if the function returns nil or all the keys have been provided as a parameter to 'user' once. When the procedure exits, each key in the table will have been provided as a parameter to 'user' once. By means of side effects the user procedure can gather statistics about, print or modify the entries in the table. For example:


```
procedure list.table( pntr t)
begin
    structure stats( int count)
   procedure print( string key;  pntr tab,  state ->pntr)
   begin
       write " -- ",key,nl
      state (count):=state (count)+1
      state
   end

   let state=stats(0)
  scan(t,state,print)
  write state(count), " entries in table",nl
end
```

The above procedure list.table causes the procedure print to be applied to the table in order to print out every key in the table.

```
procedure commit
```

This causes all the objects which can be reached from the root table to be saved in the database and then terminates the program. If the program terminates without invoking commit, none of the changes made to the database during the run of this program will take effect.

This very simple set of routines is enough to implement orthogonal persistence. Any item on the heap reachable from the entries in the root table at commit time is defined to be persistent. Although only pointers can be stored in the root table these refer to instances of structure classes. In S-Algol a structure class can be a cartesian composition of any of the types of the language. By keeping pointers to instances of structure classes it is therefore possible to make data of any type persist. It should be noted that the table handling routines are not

strictly necessary. A persistent heap requires only one distinguished pointer to act as a root for the database. A more parsimonious implementation would define the open.database routine as returning a pntr to the root of the database. In that case the only other routines required would be nil and commit. Table handling could then be an optional library.

Prototype Implementation.

The syntactic differences between S-Algol and PS-Algol are non existent. This allows the existing S-Algol compiler to be used to compile PS-Algol, providing it is given an appropriate predeclaration file including the new PS-Algol routines. The implementation of the prototype PS-Algol system thus fell into two parts:

1. Writing an interpreter for PS-Algol.

2. Implementing a Data Curator to store the persistent data.

The prototype system was designed to operate in a network environment. We were not lucky enough to have a network environment, so we simulated one on a Perkin Elmer 3220.

The Portable s-code Interpreter

S-Algol has been implemented on several machines: Z80, PDP 11, Vax. Unfortunately it had not been implemented on Perkin Elmer machines. The S-Algol compiler generates an intermediate code termed s-code. An s-Algol system must provide either an interpreter for this s-code or an s-code to machine code translator. For our implementation we decided that an interpreter would be a better initial vehicle.

We wrote our interpreter in IMP77, a high level systems implementation language [48]. It was chosen because as against assembler it is portable, IMP77 being available on a wide range of machines, and as against Pascal it allows easy access to machine level features. IMP77 allows the insertion of in line assembler in time critical loops. Our final interpreter has some dozen lines of assembler out of 1500 lines of code. These are used only in the main instruction fetch loop.

The interpreter is divided into a number of modules: the instruction fetch module, the i/o module, the initialisation module, the instruction execution module and the heap module. All operating system dependencies are concentrated in the I/O module and the initialisation module. The heap module provides storage allocation, garbage collection and all primitive operations on heap objects. These include structure field selection, vector indexing, string concatenation and run time type verification.

We verified the portability of this interpreter by porting it to Vax at a cost of 2 man days.

Architecture of the Prototype

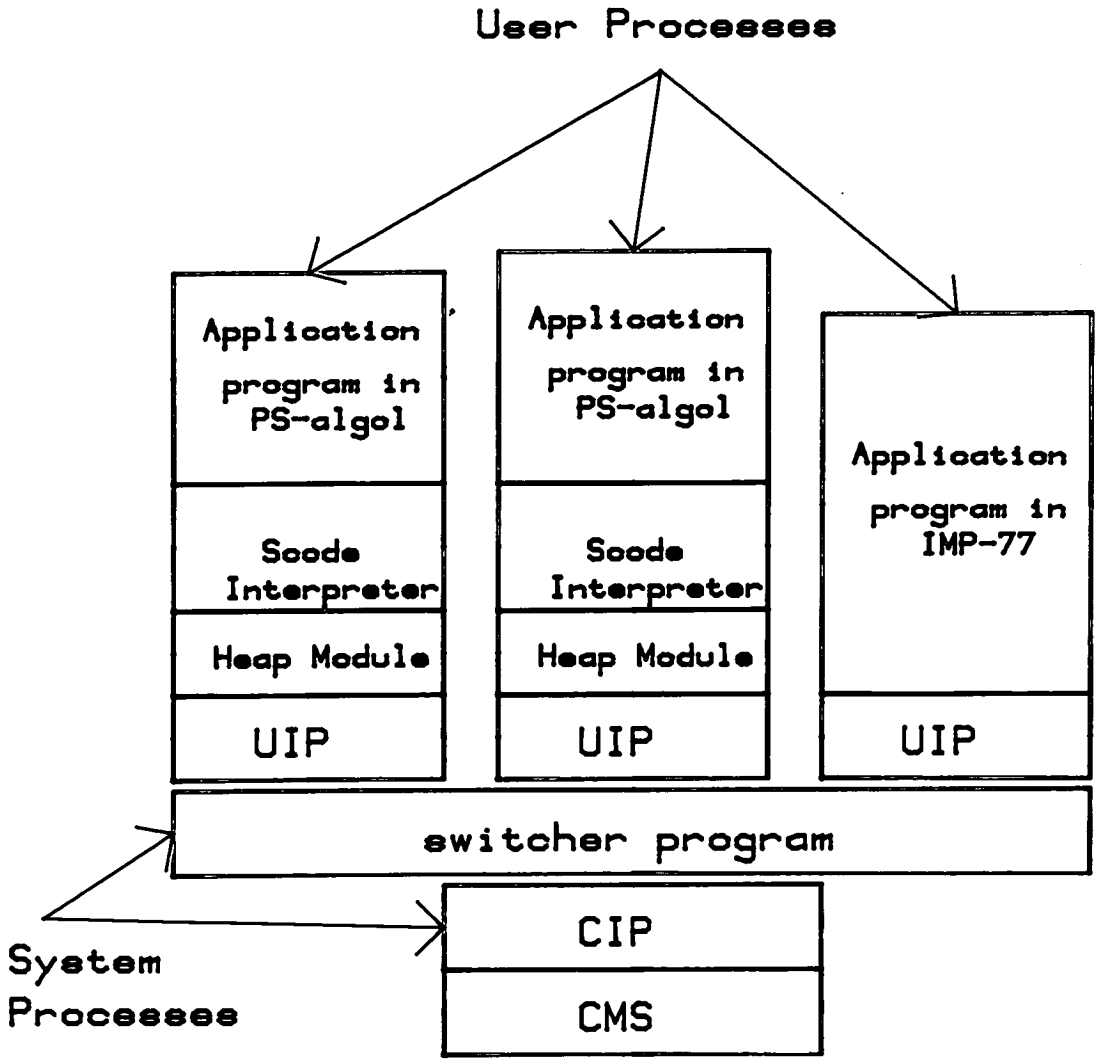

Fig 5.2

The persistent heap module.

The implementation of a run time system for a persistent S-Algol heap merely involved replacing the existing heap module with a persistent heap module, and providing a small number of new predefined S-Algol procedures. The interface between the heap package and the rest of the system was unchanged. The internal design of the heap module is dealt with later.

The Data Curator

The Data Curator is a set of software modules that provide persistent data storage and transaction management for PS-Algol. Its facilities are also used from Pascal and IMP-77 by other applications.
It provides:

1. A large persistent disk-based heap upon which untyped, variably sized chunks of data may be securely stored. It provides the operations of creating a chunk of storage, writing to that storage, reading it, extending it, and deleting it. Associated with each chunk of persistent storage is a Persistent Identifier or PID which is used as a key or address in all chunk access or update operations.

2. Subdivisions of this heap referred to as bags. Bags are named and have password protection associated with them. In PS-Algol a bag corresponds to one of the databases opened by the function open.database.

3. An associative indexing facility termed Tables. These have the same characteristics as PS-Algol tables which are implemented in terms of them.

4. A multi-user transaction facility, some of whose facilities are used to implement PS-Algol transactions. This uses the techniques described in [49]

The Data Curator is made up of 4 modules:

i. The User Interface procedures or UIP [50]
ii. The message transport facility or SWITCHER [51] [52] [53] [54]
iii. The request handler or Communications Interface Package (CIP) [55]
iv. The Chunk Management System (CMS).

The combined effect of these modules is to provide programs and processes using the Data Curator with a remote procedure call interface to database facilities that is functionally equivalent to the Ada rendez-vous construct.

The CMS is a set of routines that provide the system's storage management facilities, transaction management and table handling.
.heir functions are described in [56]. The CIP is a program that runs as a process on the PE 3220 and in conjunction with the UIP and the SWITCHER allows client processes on the same or other machines to make remote procedure calls [57] on the facilites provided by the CMS. For each service provided by the CMS there is a corresponding user interface procedure in the UIP. When called, this user interface procedure dispatches a message via the SWITCHER to Chunky. This message has the procedure encoded as a request number followed by a record containing the procedure parameters. When this message arrives at

Chunky, the CIP unpacks the message and after doing some validation checks, issues a call on the corresponding CMS routine. The result returned by this routine is then packed into another message by the CIP and relayed to the calling process where it is returned as the result of the UIP call. The SWITCHER provides the underlying transport mechanism to support remote procedure calls. In itself the SWITCHER contains no significant innovations. It is worth recording our experience with it mainly because it shows the limitations involved in attempting to construct a system of this sort on top of a conventional multi-user operating system. Secondly the SWITCHER provided a means by which we could support shared access to the data maintained by the CMS. The scheduling of requests for CMS services into a serial stream of UIP calls was ensured by the message interface.

Internal/External Transparency

It was decided that in order to postpone decisions about where processes would reside, we would use a communications protocol that was independent of the particular processor on which a curator process might happen to be executing. It should be a matter of indifference to a curator process whether the client it was serving resided in the same physical machine as itself or not.

Message Format

The Computer Science department was intending to acquire a local area communications network. This did not materialise during the research project. Nonetheless we designed our communications level on the assumption that such a network was about to arrive. The SWITCHER was designed to provide a uniform way of passing messages between processes irrespective of the machine or machines on which the processes resided. The message format as seen by programs using the message interface could be expressed in Pascal as:

```
CONST
      maxstring=127

TYPE
      netaddr= 0..65535;

      message  =RECORD
                  dest,src:netaddr;
                  body     :array  [0..maxstring] of char;
                END;
```

The messages actually transmitted over the network would contain additional fields to type the messages, which are ignored for our purposes.

The Notion of Net Addresses

The dest and src fields of messages identify the processes to which the message is directed and from which it has come, respectively. The mechanism of identification is of no concern to user processes, but at present it is provided the following way. A net address is a 16 bit integer split into two fields. The 8

3220 Switcher Architecture

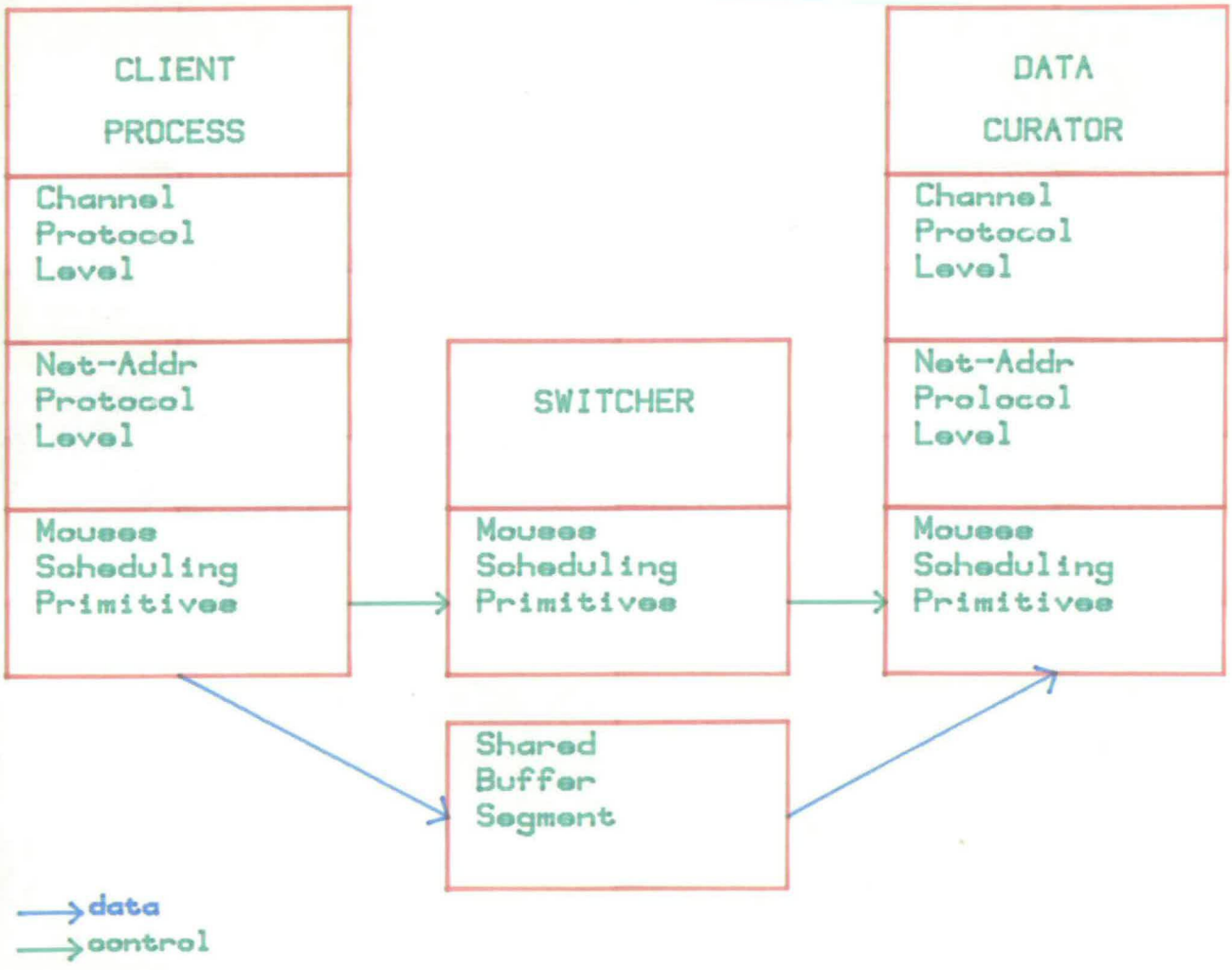

Fig 5.3

most significant bits are the machine address which will be recognised by the network station. The least significant 8 bits specify the process within the machine.

Processes within a machine may have several net addresses allocated to them, but no net address can be allocated to more than 1 process

The Switcher

In order to maintain the transparency of destination, it is necessary that on each of the processors on which the curator software runs, there should be a mechanism to ensure that messages directed at processes on the same machine get to them. So far the software has only run on two machines : a Perkin Elmer 3220 and a DEC Vax 780, and these are only linked via a point to point connection rather than over a local area net. On these, SWITCHER processes have been implemented which accept messages in net format, check whether they are directed at an net address within the set of addresses allocated to the machine on which the Switcher is running, and if so relay the message to the internal process currently owning that net address. Currently as a simplification it is assumed that the set of net addresses allocated to a machine will be a contiguous range. If the message is not directed at an internal net address it is sent into the network.

The Switchers also act as Name Servers. Processes that wish to use the network must log onto the Switcher of their machine and obtain an net address. The Switcher assigns them a free net address from the set of addresses assigned to the host machine. These net addresses are similar in function to the communication ports supported by Accent [58]

Basic Operations

The two primitive communications operations are:

wait(net address)

Which causes the calling process to be suspended until either a message directed at that net address has arrived or a timeout period has elapsed.

send(message, net address)

This transmits a message to the specified net address. The sending process can continue unless the message buffers associated with the receiving address are full, in which case it is suspended until a space becomes available.

Transport and Flow control

A communications interface of the type described, is predicated upon there being some lower level communications facility or transport mechanism to support it. For the machine to machine communication, the local area net is considered to be a reliable "black box" that will accept and deliver messages. For intra-machine communication between processes the host operating system has to

provide the transport mechanism.

On the Mouses operating system [59] message transport is effected via a common buffer segment connected to all processes using the SWITCHER. Associated with each net address within the machine is a slot in this buffer.

Send places the message in the sender's slot, and causes an operating system message to be sent to the Switcher which then copies the message onto the input queue of the destination net address. If the queue was not empty the Switcher reschedules the sending process.

Wait sends an operating system message to the Switcher asking for a message, when one becomes available it is copied into the receiver's input slot and the receiver rescheduled.

These primitives depend upon the operating system to provide a lower level scheduling system to be used in flow control. On Vax the net addresses are mapped onto Mailboxes and the standard operating system services used to transport the message.

Channels

In order to provide security, it is desirable that users should not be able to forge net addresses and pretend to be someone else. To prevent this, the sender's net address is automatically filled in by the switcher software on transmission. In order to allow a process to have several net addresses, the process is provided with process local tokens for net addresses termed channels. The mapping from channels to net addresses is carried out behind the scenes by switcher library software.

Prototype performence

The performence of the prototype was not impressive. Its performance was dependent upon three components, the interpreter, the communications subsystem, and the CMS.

The interpreter was written in a high level language and could not be expected to be very fast. Interpreted S-Algol programs on the 3220 ran at 1/8 of the speed of compiled S-Algol on a Vax 780. This was the maximum speed attained by the 3220 s-code interpreter after considerable effort at optimisation. Table 5.1 and Fig 5.4 show the improvements attained in successive versions of the interpreter. As can be seen, the improvement in performance was quite significant.

3220 Scode Interpreter Performance

The following table shows the improvement in the performance of the scode interpreter with as it was gradually optimised. Timings are in seconds.

Ver	Compile	Run
1.1	276	22
1.2	157	11
1.3	112	7
1.4	110	7
1.5	90	6
1.6	84	6
1.7	83	5
1.8	63	5
1.10	60	5
1.12	58	5
1.14	49	4
1.20	43	3
1.21	39	3
1.22	32	2

Table 5.1

It can be seen that an overall tenfold improvement was obtained. The benchmark for which these results were obtained was the compilation and execution of the following S-Algol program

Improvements in Interpreter Performance With Successive Versions

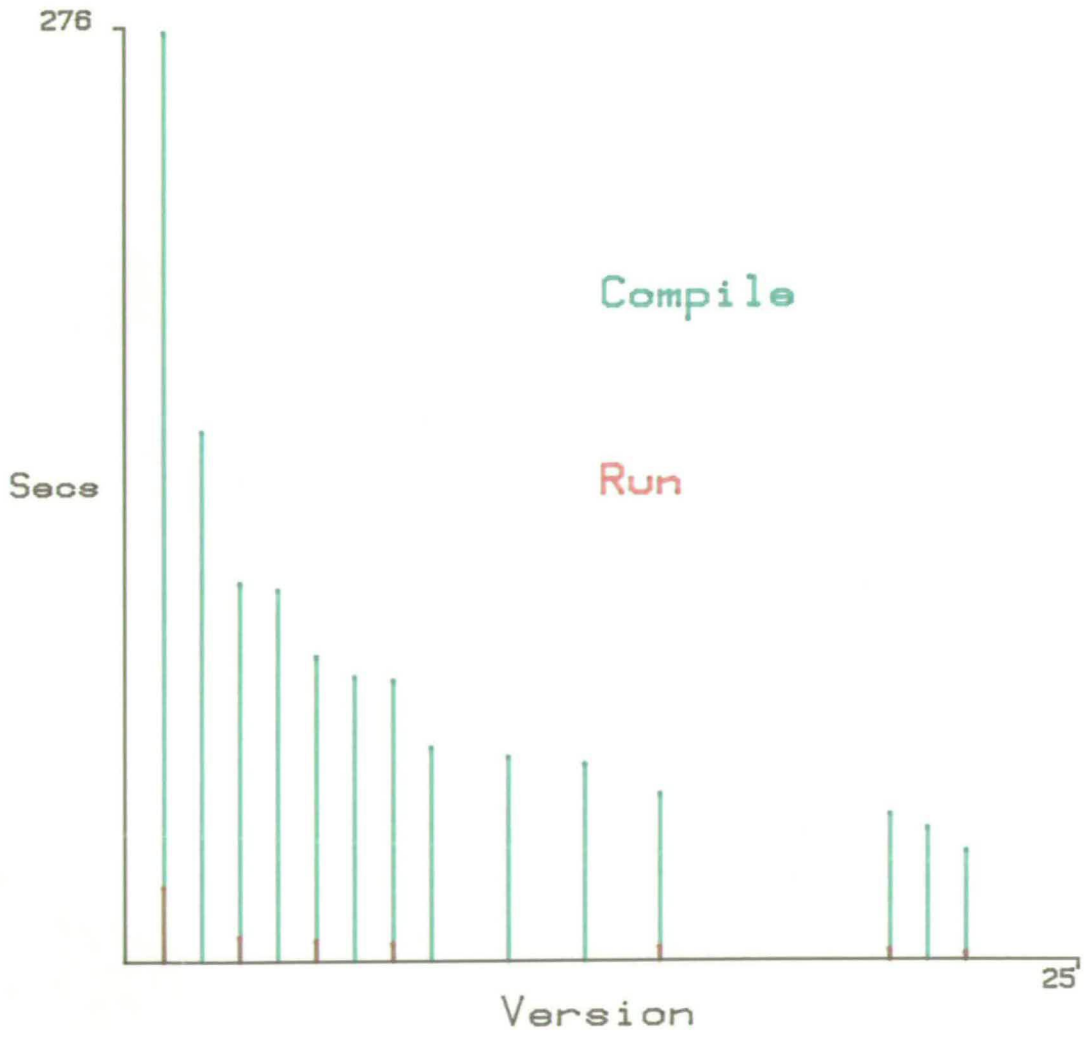

Fig 5.4

```
let array = vector 1::99 of 0
procedure squiggle (cint a -> int)   a rem 31

for n = 1 to 99 do array (n) := squiggle (n)
write "Array initialised'n"

structure list.fm (int value; pntr link)
structure NIL
let nil = NIL; let base := nil

for n = 1 to 99 do    base := list.fm (array (n), base)

let radices := vector 0::9 of nil

procedure queue.up (cint power)
begin
    let p := base
   while p ~= nil do
   begin
       let number = p (value)
      procedure ten.to.the (cint p -> int)
         if p = 0 then 1 else 10 * ten.to.the (p - 1)

      let digit := number rem ten.to.the(power); let shifter:=power
      while shifter>1 do{ digit:=digit div 10; shifter:=shifter-1}
      base:=base(link)
      p(link):=radices (digit)
      radices (digit):= p
      p:= base
   end
end

procedure reque
! takes items off radix queues and re queues them on base
begin
    for i= 0 to 9 do
   begin
      while radices(i) ~= nil do
      begin
         let trans=radices(i)
         radices(i):=radices(i, link)
         trans(link):=base
         base:=trans
     end
  end
end

for i= 1 to 2 do {queue.up(i); reque}
while base~=nildo  write base(value), {base:=base(link); '"n"}
```

Some of the more significant improvements were:

1.1-1.4 Removal of debugging and reporting facilities from the interpreter.

1.5–1.7 Replacing main interpreter subroutines with inline code.
1.8 Single character strings no longer created on the heap.
1.20 Pull and Push operations handled by inline code.
1.21 Instruction fetch loop recoded in assembler.
1.22 Use of direct access file for code loading.

Switcher Perfomance

The performance of the switcher was monitored by sending 1000 messages from one process to another. The test was carried out on the 3220 when the only active processes were the switcher, the sender and the receiver. The results obtained are shown in table 2.

	12 byte messages	24 byte messages
Elapsed time	41 secs	41 secs
Sender cpu time	0.88 secs	0.91 secs
Receiver cpu time	0.91 secs	0.85 secs
Switcher cpu time	8.15 secs	8.28 secs
Total cpu time	9.94 secs 24 %	10.04 secs24 %
in user processes		
per message	0.009 secs	0.01 secs
Total system cpu time	31.06 secs 76 %	30.96 secs76 %
per message	0.031 secs	0.031 secs

Table 2. Switcher performance for 1000 messages

It can be seen from the tables that the greater part of the elapsed time is system overhead. As all tasks were memory resident for the test it would seem that the main overhead was in context switching and rescheduling of processes. Although optimisation of the switcher code was able to produce a reduction of the switcher cpu time per message to 6 milliseconds, the effect of this was just to increase the system overhead percentage.

It was considered that the performance obtained from the communications software was not adequate. Certain phases of execution of PS-Algol programs would be anticipated to generate high volume of messages. The elapsed time required to switch these would have been prohibitively high. It was therefore decided to reimplement the system as a single process package. The CMS would run in the same process as the user program and communicate over a simple procedure call interface rather than a remote procedure call interface. The second version of PS-Algol on the 3220 and the Vax implementation followed this single process model. Nonetheless, the network server model of the CMS remained attractive for a genuinely distributed system. We were, however, able to conclude that this would demand careful attention to interprocess communication over the network with an efficient implementation of interprocess communication in the operating system kernel.

CMS Performance

The third component of the system was the chunk management system. We are not directly concerned with the design of the CMS in this thesis, but its performance as part of the PS-Algol system is of interest. Table 3 gives a summary of results obtained in a number of benchmark tests using the CMS from PS-Algol.

CMS performance part 1 10.55.04

TEST-NO	TIME	CMS-VER	BYTES	REQUESTS	CREATES	GETS
4	10.16.37	3.1	20525	1107	280	273
1	10.51.31	3.1	485	30	9	0
3	10.51.51	3.1	485	30	9	0
4	10.52.25	3.1	20569	1110	281	273
7	10.54.52	3.1	3333	92	16	24

Key
BYTES number of bytes stored or fetched
REQUESTS number of calls on CMS services
CREATES number of chunks created
GETS number of chunks fetched

CMS performance part 2 10.55.05

TEST-NO	TIME	PUTS	CRT-TABS	INSERTS	YIELDS	CMS-CPU	TPR
4	10.16.37	290	0	1	0	11036	10.0
1	10.51.31	9	0	0	0	678	22.6
3	10.51.51	9	0	0	0	520	20.3
4	10.52.25	291	0	1	0	11697	10.5
7	10.54.52	32	0	1	0	1668	18.13

Key
PUTS number of chunks put
CRT-TABS number of tables created
INSERTS number of entries inserted into tables
YIELDS number of table lookups
CMS CPU cpu time in 1/1000 ths of a second
TPR time per request in 1/000 ths of a second

Table 3. CMS performance tests

It can be seen from these tables that the time taken to service a request on the CMS varies between 10 and 22 milliseconds. The total time including system overhead, to pass the messages necessary for such a request would be of the order of 60 milliseconds. We can therefore assume that the improvement in performance gained by going from remote to direct procedure calls would be of the order of 2 to 3 times. It must be said that the performance of the prototype still left a lot to be desired.

Conclusion

The PS-Algol prototype demonstrated the technical feasibility of adding persistence to an Algol. The performance of the prototype was as much as could be expected of a first attempt, but no more, it was not usable for building serious applications on top of. It did, however, allow the exploration of problems involved in the design of such systems, such as whether they should be single process or multiprocess, what sort of heap they should have, what sort of name space should be supported. Some of these issues will be discussed in following chapters.

Chapter 6

Algorithms for a persistent heap

This chapter will present a detailed account of the design of the prototype PS-Algol heap implemented on the Perkin Elmer 3220. A second PS-Algol system was subsequently implemented on the Vax. An examination of the differences between these implementations and the rationale for these differences will be presented.

The standard S-Algol heap.

In S-Algol all compound objects: strings, structures and vectors, are created on the heap. Local variables that reference compound objects are implemented as pointers to objects upon the heap. Unlike Algol68 where it is possible to have pointers to objects on the stack or to individual fields of objects, S-Algol only allows pointers to objects on the heap and the pointer must reference the entire object. S-Algol implementations maintain two stacks. One holds all integer, boolean or real variables. The other, the pointer stack, holds all variables that are implemented as pointers.

These restrictions make the implementation of a garbage collector relatively simple. No information is required at run time about the types of variables in individual stack frames. The garbage collector merely has to preserve everything reachable from the pointer stack. This requires additional information about the types of objects on the heap, which is obtained by tagging the objects.

Object Formats

All heap objects are tagged to indicate their type. The system recognises 3 types of objects:

Structures
Strings
Vectors

Vectors

These have two header words with the format given in Fig 1, followed by the main body of the vector.

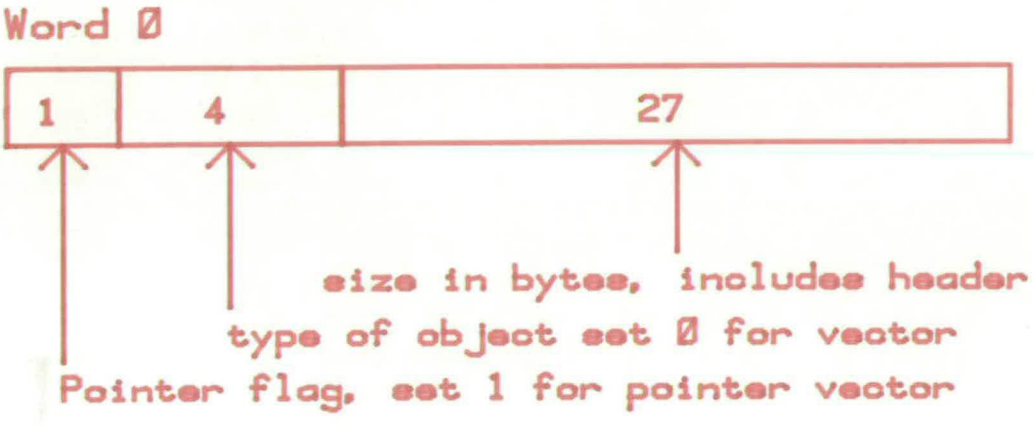

Fig 6.1 A Vector Header

Strings

These have one header word with the same fields as for a vector header word 0 except that the type is set to 2 and the size field indicates the number of characters in the string. NB this will be less than the space occupied on the heap because it does not include the header word or any packing bytes needed to maintain word alignment of the heap objects.

Single character strings have special provision made for them. At the start of the PS-Algol program, all possible one character strings are predeclared and placed on the heap. These are then accessed via an array of pointers termed the string table. This significantly reduces the number of calls on the space allocator to create strings. This in turn reduces the frequency of garbage collections.

Structures

These have a two word header. The first word is like that of a vector except that the type is set to 15. The second word is the "trademark". Each structure class has a distinct trademark. The trademark is used to index a "structure table" which describes for each class:

Its size in words.
The number of pointer fields it contains.

The pointer fields, if any, of the structure are concentrated in contiguous locations at the start of the structure. The garbage collector thus has sufficient information to find the objects referenced by an instance of a structure class

Local Object Addressing

Consider that the heap is divided into two portions, the local heap which is currently resident in main memory and the global heap which is in the database. All addressing of local heap objects goes via a structure called the Persistent Identifier to Local Address Map (PIDLAM). This is a table indexed on Local Object Numbers, that stores for each local object number the address on the heap at which that object is stored, the database address or Persistent Identifier of the object, and two flag bits termed the LI flag and the GC flag.

The PIDLAM is optimised for mapping Local Object Numbers to heap addresses on the assumption that most references in the course of a program will be via Local Object Numbers. However, it is sometimes necessary to dereference Persistent Identifiers. This can happen under two circumstances:

1. A pointer returned by one of the interface procedures is being dereferenced.

2. A pointer field of an imported object is dereferenced.

```
structure TRIPLE( string s;  int i;  *int v )
```

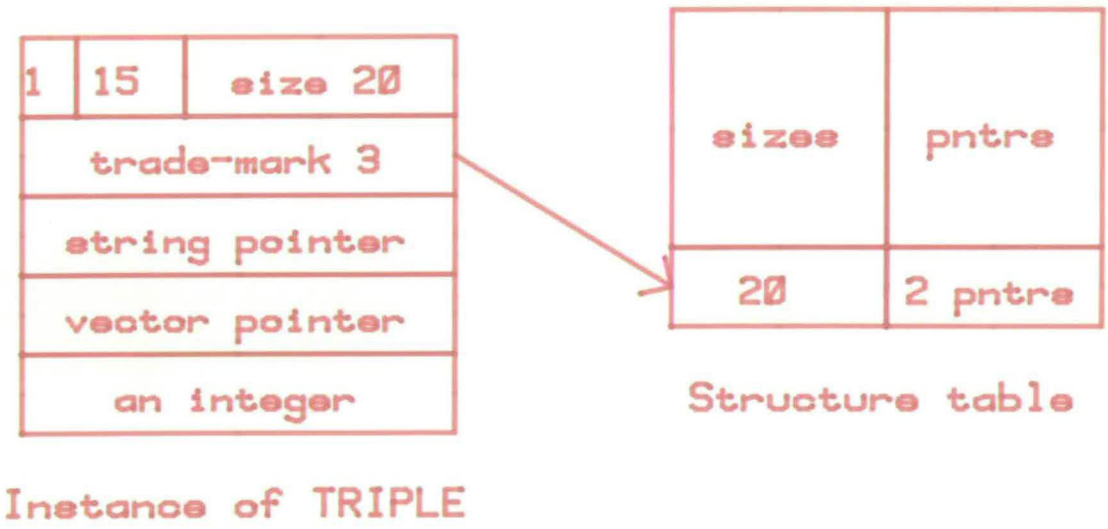

Fig 6.2 Structure Class Representation

In either case an attempt is made to look up the PID in the PIDLAM. If the PID is found the corresponding local address is returned. If it is not present in the PIDLAM the object is loaded into the heap from the database and an entry made in the PIDLAM associating a new local object number with the PID and the address of the item on the heap.

Storage Allocation and Recovery

The local heap is managed by a compacting garbage collector.

Garbage collection may occur for either of two reasons: because heap space has been exhausted or because the PIDLAM is full. In the event of garbage collection not satisfying the request for space, the entire local heap is returned to the data curator.

As with S-Algol the interpreter maintains two stacks, the main stack which holds all integers reals and booleans, and the pointer stack on which all variables referencing strings, vectors or structures are located. Obviously, everything reachable from the pointer stack must be preserved during garbage collection. The pointer stack is not the only root for the garbage collector, in addition all the one character strings on the string table must be kept. Also since we are maintaining a persistent heap, all the objects brought over from the database and all the things on the heap that are reachable from them must be kept. The LI flag in PIDLAM entries indicate whether an object is Local or Imported. All objects whose LI flag is set are database objects that have been imported. The garbage collector can be characterised by the following notional algorithm:

```
procedure collect(int space.required)
begin
        mark all reachable from pstack
        mark all reachable from imports
        mark all reachable from string table
        recover and compact free space
        if free space < space required then
        begin
                send all marked items back to database
                convert pstack to PIDs
                reinitialise PIDLAM
        end
        else
                unmark all objects
end
```

The marking is done by a recursive routine in a fairly straightforward way:

PICS C6FIG3

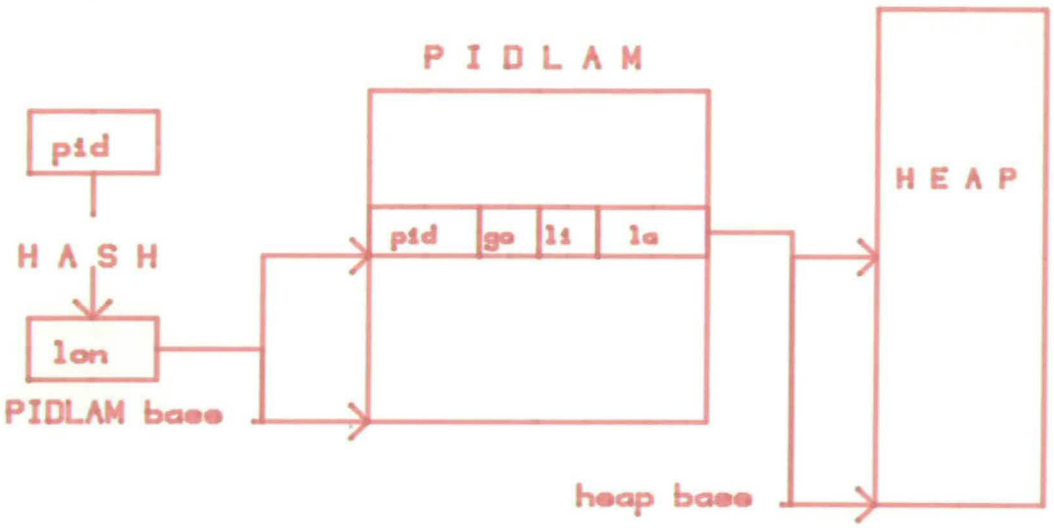

Fig 6.3 Addressing Via the PIDLAM

```
procedure mark(pntr p)
begin
        unless p is PID or PIDLAM(p, gc) then
        begin
                PIDLAM(p, gc):= true
                store p at end of item
                if pointer bit of item do
                        for all pntr  in item do mark (pntr)

        end
end
```

The only unusual features of this marking algorithm are the use of the PIDLAM to hold the mark bits, and the storage of the item's LON immediately after it in RAM as shown in Fig 4. This is an optimisation for use in the compaction phase. Compaction is achieved by a linear scan through the heap that copies down all the marked items and simultaneously updates the local address fields of their PIDLAM entries. This requires that having found a marked object on the heap, we can find its PIDLAM entry. As the PIDLAM is only optimised for access via LONs, this patching of the LON at the end of each marked item enables updating of the local address fields to be faster. When objects are created or moved down, space is left to hold their LONs during the marking phase. The space is initialised to zero, since LONs are all non-zero it is possible to distinguish a marked from an unmarked item given its local address.

If the garbage collector runs out of space either on the heap or in the PIDLAM it sends everything on the heap back to the database. The algorithm that does this has the following notional structure:

```
procedure send all back
begin
   for all lons in PIDLAM do  begin
      if lon is used do
         if marked do
         begin
            for all pntrs in item do
               unless nullfile, nilstring or 1 char string do
                  if pntr is lon do   convert to pid
            send back item
         end
   end
end
```

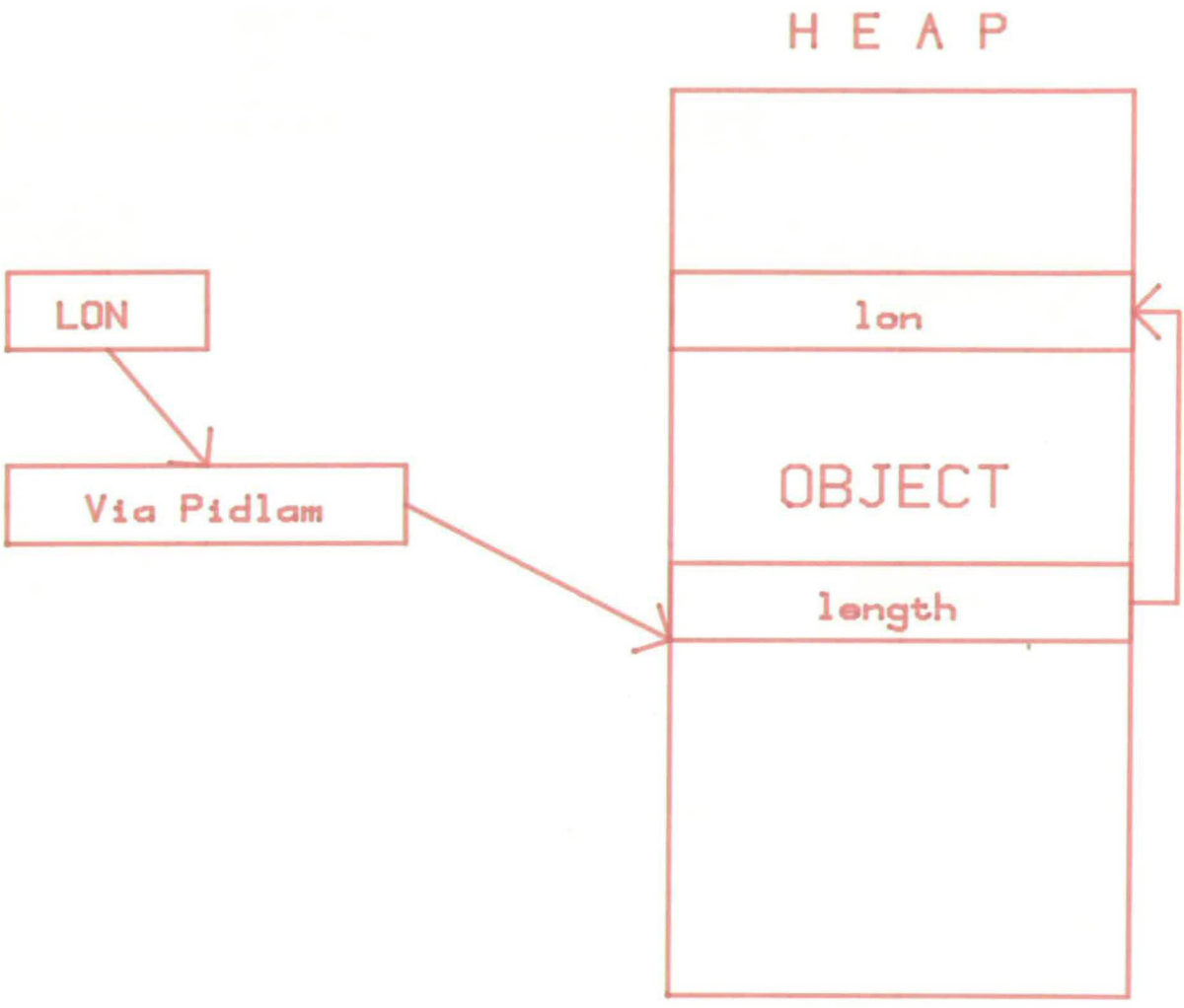

Fig 6.4 An Object After
Being Marked

Committing a Transaction

A program in PS-Algol can terminate in one of three ways:

reaching the end of the program
executing an abort statement
executing the commit routine

In the first two cases the run time system signals an abort transaction to the database system and all changes made to the database since the start of the program are invalidated. The implementation of this rollback is very efficient. In the final case, the program must go through a more elaborate close down sequence. Whilst the program has been running the storage management system has been maintaining 2 lists referred to as the N list and the O list.

The N (for New) list is a list of the new Pids created during the run of this program. The O (for Old) list is a list of the Pids of all of the items brought over from the database in the course of this run of the program. These items are themselves persistent, so it is assumed that anything reachable from them must also persist. As an extension of this, any item inserted into a table is assumed to be persistent and is put on the O list. Because the garbage collector may have run out of space and sent the entire heap back one or more times, the N list may contain items that are not reachable from already persistent items. These items, which have been given a database identifier temporarily, must be deleted from the database as they will be unreachable from the database root. The strategy followed is to delete from the N list all the items which can be reached from the O list, and then delete the items that are left in the N list from the database. The removal of these temporary items is called 'threshing'.

The close down algorithm is complicated by the fact that the O and N lists may get to big to keep in RAM. We therefore maintain the lists on the database as a linked list of database records, each of which may hold 20 pids. However, the algorithm would run prohibitively slowly if the lists remained on disk whilst it ran. We have therefore compromised by first emptying the heap onto the database, then loading the lists into the space freed in RAM. What is left of the heap space is used as a heap for the evaluation of the algorithm, since the marking phase of the algorithm requires the use of the heap. Sending everything back before threshing has the advantage of making sure that all items that are candidates for persistence are assigned PIDs. We can therefore present an outline algorithm to carry out this close down:

```
procedure close
begin
   mark all reachable from imports;
   send all back;
   pull N list into RAM;
   initialise heap with remaining store;
   thresh;
   commit transaction;
end
```

The procedure thresh does the sorting of the wheat of items reachable from the database root from the chaff of temporary variables and has the following outline:

```
procedure thresh
begin
   ! mark all referents of O list items
   while  O.list.head ~= nil do begin
      ! loop A
       mark referents in N ( O.list.head (pid))
       O.list.head :=  O.list.head (next)
   end
   ! now mark items in N list recursively reachable from
            marked items
   let  number marked:=0
   while
       ! loop B
       let n:= N.list.head
       while  n ~= nil do begin
          ! loop C
          if marked (n) do begin
              number marked plus 1
              mark referents in N (n)
              ! remove it from the list
              n(next):=n(next,next)
          end
          n:=n(next)
       end
       number marked > 0
   do  number marked:=0
   !  delete the remaining items on the N list from the database
   while N.list.head ~= nil do begin
         ! loop D;
         delete (N.list.head(pid))
         N.list.head:=N.list.head(next)
   end
end
```

Note that the recursive marking of the items reachable from the N list is performed by a widthwise rather than depthwise traversal of the reachability tree. This is done because it reduces the traffic to and from the disc. Items are fetched into memory one at a time and then their pointer fields are compared with the pointers on the N list which is held in memory. If we carried out a depthwise traversal of the tree there would be the danger that the number of items on a sub tree would be so great that the heap would fill up whilst it was being traversed, which could lead to the root node being swapped out before all its dependents were marked. With a traversal of the type we have chosen, each item referred to on the N list is brought into memory from disk only once.

Weaknesses of the Algorithm

The algorithm is limited in the size of transaction it can handle. If the number of new items created in the course of executing a program grows too great the N list will not fit into memory and the transaction will, in consequence, be unable to commit.

The algorithm is optimistic. It preserves all items which could be persistent. It will preserve some items which are not reachable from the database root. It fails to detect when some of the already persistent database becomes unreachable due to a change in the chain of persistent references leading to it from the database root. The algorithm only threshes out newly created items that are not reachable from the database root. The database will thus tend to accrue wasted space. This can only be resolved in the end by doing a garbage collect of the entire database.

The algorithm is also rather slow. Its exact performance will depend upon the proportion of items on the N list that are on the trees of reachable items growing out from the O list and upon the length of the paths from the O list to these items. If we assume that marking of items on the N list is done by marking the list elements rather than the items in the database that these point at, then the procedure that marks all referents of an object in the N list will have a time penalty of:

$$kN + F$$

where
N is the number of items on the N list
k is the number of pointers in the root object
F is the cost of fetching the root object from the database.

Thus the cost of loop A will be:

$$O(KN + F)$$

where
O is the number of items on the O list
K is the average number of pointers in an item

The cost of loop B will be determined by the cost of loop C times the maximum depth of the tree from O list to the farthest reachable branch on the N list. The cost of going round loop C for the i th time would be :

$$M_i (KN_i + F) + N_i - M_i$$

where
M_i is the number of marked items on N list for ith iteration
N_i is the number of items on the N list for the ith iteration

For small N this will be dominated by the disk transfer time to fetch items from the database, for larger N it may become dominated by the

time to scan the list. The total cost of loop B is thus:

$$\sum_{i=0}^{d} M_i (KN_i + F) + N_i - M_i$$

where
d is the maximum depth of the reachability tree

But $N_{i+1} = N_i - M_i$ so that the cost of d iterations reduces to :

$$d\left(m\left(\frac{N}{2} + F\right) + \frac{N}{2}\right)$$

where
m is the average number of items marked per iteration.

If we consider the worst case, where all items are eventually found to be reachable from the O list and must therefore be kept, we obtain an upper bound on d:

$$d <= \frac{N}{m}$$

This gives an upper bound for the cost of loop B of:

$$\frac{N^2}{2} + FN + \frac{N^2}{2m}$$

The cost of the final loop will be:

$$N_{d+1} D$$

where
D is the cost of deleting an item from the database.

Loop B is clearly the most costly unless the number of reachable items on the N list is very small. In the worst case this algorithm is of order N^2. If the structure of the N list was changed to a binary tree the N^2 terms would be reduced to N log N. At this point the linear component of the cost due to fetches from the database will be the dominant cost. If we assume that the cost of deleting an item from the database is approximately the same as a fetch, then the sum of the costs of loop D and the linear component of loop B will be independent of the fraction of the new items that are to persist.

For interactive use the delay on committing a transaction is irritating. This could be reduced if the work of evaluating the close down algorithm was farmed out to a machine acting as a database server where it could run as an asynchronous process after the heap had been cleared out.

Vax Implementation

The pertinent difference between the Vax and the PE3220 is that the former has a large virtual memory space, whereas the latter has a much smaller segmented memory space. A basic assumption made in the design of the Vax implementation was that the heap could be made large enough to greatly reduce the risk of the heap or PIDLAM filling up. If the heap never fills up, then it never has to be sent back in the course of the program, only at the end when a commit occurs. This makes it unnecessary to go through the old rigmarole of N and O lists. At commit the system must merely discover which things on the heap are currently reachable from imports. This can be done by chasing down the PIDLAM recursively marking everything whose imported bit is set.

The Vax implementation also differs in the actions taken when an item is imported from the persistent heap. In the 3220 case the item was copied over unchanged from the database to the heap, a PID was assigned to it, and a PIDLAM entry made. In the Vax case, in addition to assigning a PID to the item, all of its pointer fields are converted from PIDs to LONs. This may involve the allocation of LONs to items that have not yet been brought into the heap from the database. This has the advantage of making access to imported items faster, as all items are accessed via their LONs, and LON access is done by a simple indexing operation whereas PID access involves a table look up.

The added speed is bought at the usual price: space. But on Vax we have spare space to pay for our speed. The extra space requirement arises because the PIDLAM grows faster, having to have entries for items not yet on the heap.

For any finite sized heap there will still be a risk of its overflowing in the course of a program. One can take the attitude that this happens in any system with a heap, and that the best you can do is make the heap big so that it is rare. In any case, the database cannot be damaged by the heap overflowing and the program aborting, since the transaction will have been abandoned and no changes will have been recorded in the database. Still, it is possible to reduce the probability of the heap overflowing by taking simple prophylactic measures. There are three overflow scenarios to be considered:

1. A program builds up a huge temporary run time data structure which exceeds the size of the heap.
2. A program goes through a large portion of the database and changes it. If the database is much bigger than the heap the heap will overflow.
3. A program scans the database for report generation, or statistical purposes, touching but not modifying much of the data.

The first case can only be solved by having a bigger heap. It is identical to the problem that arises with an ordinary non-persistent heap if you try to use too much memory.

The second example can be solved by keeping a changed bit in the PIDLAM which is set whenever an item is changed. Then when you run out of space you throw away all the unchanged imported items on the heap. It will always be possible to recover them from the database if they are required. This has the problem however, that in the Vax model of addressing the PIDLAM entry for the thrown away items must be retained. This is because there may still be items on the heap containing LONs that point at the discarded items, and the PIDLAM entry corresponding to that LON must be kept.

The solution to this problem enables us to deal with the third example as well. It uses the following algorithm:

```
mark everything recursively reachable from imports
 for all lons in PIDLAM  do
   if marked (lon)  do
       if changed (lon)  then send.back(lon)  else discard(lon)
clear mark bits
mark everything recursively reachable from P.stack
 for all lons in PIDLAM  do
     unless marked (lon)  do
       unless present (lon)  do
         delete lon from PIDLAM
```

The effect of this is to leave on the heap only local items. The PIDLAM will contain entries only for those items reachable from the P.stack or from local heap items, but will contain no entries for items only reachable from persistent objects.

Comparisons

The mechanism used for the persistent heap on the Vax implementation, along with the improved algorithm to handle heap overflow (not yet tested) seems to be the best way of supporting a persistent heap so far examined. It is likely that future machines will all have big enough virtual memory spaces to support it. The 3220 approach suffers from a slow close down, and slow accessing of even heap resident imported items.

Another approach reported in the literature, the Small-talk virtual memory system, accesses all items whether persistent or not by means of their PIDs (termed object pointers in Small-talk). This means that all pointer accessing has to be done by hashing into the Resident Object Table (Small-talk for PIDLAM). In the Small-talk system pids are not converted into lons when an object is loaded onto the heap.
When running on special hardware this hashing can be speeded up by using microcode, but for a given hardware configuration this will always be slower than the indexing of the PIDLAM used for all accessing on Vax PS-Algol. Let us assume that the accessing algorithm is written in micro-code. Then the cost of an access method can be approximated by the number of memory fetches required, on the assumption that these are the rate limiting step of the process. Let's look at the costs involved in the two methods:

STEP	VAX-PS-Algol	SMALLTALK
Hash pid	0	0
Look up using hash	0	1.5 approx
Fetch local address	1	1
Access field	1	1
Total cost	2	3.5

Another possible approach would be one based upon a conventional paged virtual memory system. In this the cost of finding an object is divided into the cost of finding the page in which the object is located, and then the cost of fetching the field of the object from within that page. The cost of finding the page and finding an object is the same provided that they are both resident. In each case it is an indexing operation. The difference arises in the probability that an object will turn out to be resident. In the case of a traditional paging mechanism this probability falls as the ratio of virtual to physical memory rises, unless you have good locality of access. In the case of a heap this locality of access is difficult to ensure. In PS-Algol the probability of residence is determined by the ratio of physical memory to objects touched in this program. On the assumption that the set of objects touched will be a subset of the total database, it can be hypothesised that the probability of finding an object in memory will be better in PS-Algol than in a traditional paging system. Confirmation of this assumption would have to be made by means of a controlled experiment involving the construction of two PS-Algol systems running on a machine with paging facilites, one of which used these and the other of which used our algorithms. Comparative tests of the two methods would then be possible.

NEPAL

As was mentioned earlier,. the language PS-Algol was conceived as a step on the road to greater things. Our earlier research into ways of providing persistence for programming languages had led to the conclusion that the incorporation of persistence required a number of structural changes to languages if full advantage was to be taken of it. The language NEPAL was designed in the light of these conclusions. PS-Algol was to be the way to bootstrap NEPAL.

The experience of building PS-Algol showed that in practice it was possible to set up a persistent Algol with far fewer changes to the original language than we had at first suspected. However, in the light of our original aims, PS-Algol can be seen to be deficient in a number of respects. This is not to deny that it is a very useful system in its own right. It does implement persistence as an orthgonal property. Data of any type may be made to persist so long as it is reachable from the root table. Such experience as we have with it so far, indicates that it does make it much easier to write database type software. Indeed it is proposed to make it a public product, and to carry out serious investigation into the productivity that can be attained using it to implement a number of database systems. But on a-priori theoretical grounds there are reasons to suspect that a number of weaknesses will be exposed in the system.

PS-Algol Weaknesses

1. Inhomogeneous Scope Rules. The table facility can be considered as providing a new means of declaring variables. These variables are in an outer persistent scope that is common to all programs using a given database. It differs from other forms of declaration in three respects:
   - i. variables are declared at run time not compile time;
   - ii. variables can be deleted from a scope, as well as introduced into it;
   - iii. the declaration facility is nonorthogonal over types, in that the only type that can be declared is type pointer. Of course it would be relatively easy to introduce a set of routines corresponding to enter, lookup etc, for each of the base types of the language. This would be aesthetically unappealing, besides failing to solve the problem. Thanks to the vectoring operation over types, an infinite set of types can be generated within S-Algol by successive application of this operation. We could by providing additional routines, go only a small way towards covering this infinite set.

   Of these features, the first two can be considered desirable in a persistent language, the latter can not. Why are they desirable ?

   Consider the normal process of program development. In this programs are weighed in the balance and found wanting, so they are

modified and put once more to the test. When considered satisfactory they are put into use. After a while it is discovered that some further features would be desirable. It is modified again, more variables are added and more procedures to act upon them. If we consider this in a persistent context, it is clear that we will want to declare new identifiers in a persistent scope after it has been created and has been in use for some time. This implies some form of dynamic declaration facility.

2. Poor Sharing. The facilities for sharing data are very primitive. Data can be shared between programs by running them against the same database. But they must be run seqrentially against the database. Simultaneously running programs may not run against the same data.

3. Poor Facilities for Data Abstraction. This is one of the weak points of the original language S-Algol. It provided no means for the declaration of abstract data types. The representation of a structure class can only be hidden via the normal Algol scope rules, i.e., by declaring it inside a procedure or block. However, this means that only one procedure can act upon that structure. In the Vax version of PS-Algol some extensions to the scope rules have been made to improve on this [60], but these are still limited. It needs hardly be emphasised that data abstraction facilities are likely to be important in data intensive persistent programming. The fundamental weakness of the data abstraction facilities stems from the next point:

4. Procedures Not First Class Citizens. To provide good data abstraction facilities you need to be able to tightly bind data to the code that acts upon it. If you have persistent data you should have persistent procedures bound to it. This implies having procedures as objects on the persistent heap, rather than as parts of a program file that is outside of your database system. There are other advantages to making procedures objects, in that you can then construct higher order functions, but this was not really considered when designing NEPAL.

5. Compiler Not Interactive. We observed in an earlier chapter that insofar as persistence has been incorporated into programming languages, it has been interactive languages like Lisp, APL and Smalltalk that have taken the first steps. If one is implementing a persistent programming system, it seems highly desirable that it present an integrated view of programs and data. At one level both of them are persistent data structures. The program should be kept in the same heap as the data. The notion of compiling a program is a hangover from the days of cards which had to put through the computer to be compiled. What one wants is to have a persistent store containing a structured collection of functions and data. Editing a function can then be considered as the same type of operation as modifying any other part of the database such as a structure field, a tuple of a relation etc.

6. **Data Definitions Not Updatable.** Just as programs are never right, neither are data definitions. Sooner or later you want to change them. You want to add new fields to structure classes for instance. PS-Algol does provide an updatable data type in the table. A table can be treated as a record with strings as its field names. Data structures using tables can readily be redefined to have additional fields, but suffer from the inefficiency of having to do all accesses by tree search. It would be desirable to be able to modify the definitions of structure classes in order to add new fields to them. This gets you into the whole area of database reorganisation. It should be possible to provide for this in the language design.

These were some of the design problems NEPAL was intended to solve. In what follows examples are given of NEPAL syntax. Because of the restrictions of the mode of presentation in a printed text, these are made to look like ordinary program texts for a batch programming system. It was intended however, that NEPAL function and class definitions be considered as data structures that could be presented to the programmer in a number of ways other than the simple textual form. Program syntax is a method of mapping a graph structure onto a linear sequence of characters. In a persistent programming system the program can be held as a graph, and displayed in a variety of ways. The form of presentation given here is highly provisional and should not be taken as defining that used in an actual NEPAL system.

Groups

A Group is a persistent scope that holds a set of identifiers. These identifiers may be of any one of the base types (int, real, string and pntr), or may be vectors, classes, procedures or other groups. Identifiers are introduced into groups by the normal S-Algol forms of declaration. There is a form of declaration to allow for the creation of new groups.

<group.decl> ::= group <identifier> <- <identifier.list>

This declares a new group with the specified identifier, from which the identifiers in the identifier list will be visible. The identifiers in the identifier list are referred to as the imported identifiers, as they are imported into the scope of the new group. The imported identifiers must currently be in scope.

Group level commands

The Nepal compiler would be interactive. On initiation the compiler is in the context of the distinguished group TOP.

Any valid Nepal statement typed by the user will immediately be executed. The set of valid statements is specified by the following syntax:

```
<statement> : :=<declaration>
              | <class.decl>
              | <group.decl>
              | <clause>
              | be <identifier>
              | forget <identifier>
              | get <filename>
              | quit
              | stop
              | view <identifier>
```

The be statement takes as its parameter a groupname that is currently in scope and causes that group to become the current scope.

The quit statement causes the group enclosing the current scope to become the current scope.

The forget statement causes the identifier that it takes as a parameter to be removed from the current scope.

The get statement causes the file that it takes as its parameter to become the current compiler input stream. On encountering an end of file the current input stream reverts to the previous one. These get statements may be nested.

The view statement causes the type of the identifier it takes as argument to be printed.

The stop statement causes compilation to cease.

```
group scrabble
  be scrabble
    class letter.constants (
      let a = 1
      let z = 26
      let letter.score = @ 1  of [1,4,3,2,1,3,2,4,1,
                                  8,3,2,1,1,1,1,10,1,1,1,1,3,4,10,4,8]
  )
    class word (
     let txt=  default " "
     let next:=  default nil
   ->txt,next)
    class utilities (
      archetype letter.constants
      proc a
      proc b
    . . .
  )
    class lexicon (
      archetype letter.constants, utilities
      import word                          !words in the language
      let vocab =  vector 1::15  of         !index on length
                   vector a::z  of nil     !then on first letter
      proc find.word (string s ->  pntr)
           {.....}
      proc insert.word (string s)
           {...  abort. ..}
      trans add.word (string s->  default  false)
           {.....}
      trans add.list.word (string file)
           {.....}
      proc find.all (string template,using ->  pntr)
           {.....}
      proc print.words (string query)
           {.....}
     ->  add.word, find.all, add.list.words, print.words
  )
    let English = lexicon; English(lexiconadd.word)("fish")
 quit
stop
```

SESSION 1 - CREATING A GROUP

Example 7.1

Example 7.1 shows a sequence of text which creates a new group called "scrabble". It is made to possess 5 identifiers: "letter.constants", "word", "utilities" and "lexicon" which all identify classes, and "English" which identifies an instance of the class "lexicon". The internal structure of these classes and the statement adding the word "fish" to the "English" language are explained below.

```
be scrabble
  English(lexiconadd.list.words)("Chambers")
   let American=lexicon
  American(lexiconadd.list.words)("Webster")
   class board(. ...)
   class tile.box (. ...)
   class game (
      import lexicon, tile.box, board
      let language:=nil
      let tiles    :=nil
      let board    :=nil;  let my.turn:=  false
      let  started:=  false;  let my.score:=0;  let your.score:=0;
      trans start {....}   !choose language then play
      trans resume {....}  !continue play
     -> start, resume)
   group play <- game
   be play
      let g = game
   quit
quit
```

SESSION 2 - EXTENDING A GROUP
Example 7.2

The example session shown as Example 7.2 illustrates the persistence of the data structure for "English", and shows a new instance of a lexicon being created. Both are populated with vocabularies. Two more classes are created then the class game is created to record the current state of play. It provides two operations on a game. They will use operations on lexicons already provided. Note that many instances of game may exist simultaneously, each referring to and operating upon one common lexicon, as various games proceed in the same language. Each application of "start" or "resume" on each one of these games is analogous to running a transaction (such as a CAD design step) against a database composed of a specific part and a common part. Thus aspects of concurrency, transaction and structured databases are accommodated.

Finally a subgroup "play" of this group is created. It only has an instance of a game in it. Users of this group would therefore, only be able to play the game with start and resume, and would not be able to pry into or amend the vocabulary of the language they were playing. This illustrates some of the protection mechanisms available. It is now appropriate to look more closely at the provision of classes.

Classes

Classes perform a number of functions in the language. They provide:

i) A unit of design of data structure and program. They may be designed independently, or may be composed from one another. Two composition rules are possible. A class may import identifiers including the names of other classes so that it may reference or create instances of them. A class may also be the concatenation of its list of archetypes plus the fields defined within that particular class. This provides in one mechanism both aggregation and specialisation.

ii) An association between data and function. The only data which a procedure, transaction, or initialisation code may operate on, is data instances of the class in which that code is defined or in the archetypes of that class. Similarly the only code which may directly operate on an item of data is code associated with the class of which that data is an instance.

iii) A basis for data structure description. The fields within a classes define all the data which may exist, and the procedures within the class define its semantics and constraints.

iv) A unit of data generation. Data may only be generated by creating an instance of a class. The class name invokes the generator. The class definition rules are so arranged that this ensures that all data elements are initialised.

v) A logical unit of data locking. Whenever a transaction is invoked, the referenced instance is automatically locked to prevent other transactions interfering. This means that data only accessible via that instance is also made accessible only to the one transaction. Hence the data designer has the possibility of arranging economic locking strategies. (In the example above update operations on words (insert words) need not be designated as a transaction, since a word can only be in one lexicon and updates only occur within transactions on a lexicon (add.word,add.list.word)).

vi) A unit of compilation. The environment imported by a class is entirely defined within the class. Similarly its exported environment is defined.

vii) A program or constant library. If a class has no variable fields then when used as an archetype it costs nothing in the storage of instances, but provides a library of procedures, transactions and constants. This is illustrated in the preceding examples where the classes "letter.constants" and "utilities" are set up for this purpose, then used in the class "lexicon".

A class has the syntactic form

<class.decl>::= class <identifier>(<S.spec>)

<S.spec>::=[<externals>][<sequence>][<arrow><exportlist>]

<sequence>::=<declaration>[;<sequence>]
|<clause>[;sequence]
| <empty>

<externals>::=

[archetype <typelist>]
[import <identifier.list>]

e.g.

```
class person(
   let name= default "unknown"
   let sex = default "female"
   let spouse:= nil
   trans marry( pntr partner) {. .... }
   proc spouse.of(-> pntr) spouse
-> name,sex,marry,spouse.of)
```

The above declares a structure class person with 5 fields, two of which are constant strings, one of which is a pntr variable, one of which is a transaction and one of which is an ordinary procedure. this expression yields a result of type pntr. Each declaration declares a field:

let <identifier> = <expression>
declares a constant field of the type yielded by the expression. Special expressions of the form.

default <expression>
indicate that the initialisation expression is merely a default that can be redefined at instance creation.

let <identifier> := <expression>
declares a variable field in a similar way. The class "word" in the preceding example has both a constant and a variable field.

proc <identifier> <parameter pack> <body>
declares a procedure field. Examples shown are all the fields in "utility" and find.word, inset.word and find.all lin "lexicon".

trans <identifier> <parameter pack> <body>
declares a transaction. A transaction is identical with a procedure except for locking and completion mechanisms which are defined below.

Instance Creation

The S-Algol syntax for creating an instance of a class has been extended to allow for the default initialisation of fields. In S-Algol, all fields of a structure had to be provided with initialising values at creation time. In NEPAL it is not necessary to explicitly provide initialising values at instance creation as all fields have default values declared at class definition. The default values can, however, be overridden.
So we could say:

```
let Ms.X=person
```

or

```
let Mr.X=person(sex=male)
```

or

```
let John.Doe=person(sex=male, name="John.Doe")
```

Archetypes

The notion of archetypes permits the construction of semantic hierarchies. A class inherits all the fields of its archetypes. Thus in the class student:

```
class student(
   archetype person
   let year.of.enrolement = default 1980
   let year.of.study := 1
   let course := nil
   proc accommodation.requirement(-> string)
      if spouse = nil
      then "single room"
      else "double room"
-> year.of.enrolement, year.of.study, course,
   accommodation.requirement)
```

the spouse field of the class person is available. Further, all instances of the class student will also be instances of the class person.

e.g.

```
let John=student
```

Then the expression:

```
John is student and John is person
```

would yield the result true.

A class may have several archetypes. An instance of a class is an instance of each of its archetypes. This "is a" relationship is transitive.

If a student is a person then any code that operates on persons must be able to operate on students. This in turn implies that the exported fields of person must be available:

e.g.

```
John(person . name):= "John"
```

is permissible.

Indexing

In Nepal all structured objects whether instances of structure classes or vectors exist on the heap, and the generators for these objects yield pointers to them. To obtain the elements of a vector or the fields of a structure indexing is used. For a vector, the index is an integer.

< name >(< clause >)

Before any indexing is performed the bounds of the vector are checked against the index for legality.

In the case of structures indexing is done using one of the fieldnames mentioned in the exportlist. However, since several structure classes may have fields with the same names it is necessary to prefix the fieldname with the classname, to enable the class of the structure to be checked. This is a change to the S-Algol

naming convention which made the fieldnames of structure classes simultaneously in scope unique.

<name> (<identifier> <identifier>)

The language also provides the binary operators is and isnt for checking whether or not a pointer refers to a member of a particular structure class.

Class scope rules

The introduction of classes involves certain modifications of classical Algol scope rules. Within the body of a class the following names are visible:

i) the names declared earlier in the body of the class;

ii) the names declared in the bodies of the classes mentioned in its archetype list;

iii) the names explicitly introduced via the import list.

In the classical scope rules a name may be redefined in an inner scope so that the new definition hides the previous one. A similar rule applies with archetypes. A name defined within a class hides any definitions of the same name within its archetypes. By extension a definition of a name within the body of a class occurring earlier in the archetype list is hidden by a definition of the same name within the body of a class that occurs later in the archetype list.

Note that a *pntr* may be passed about or stored as if it were an untyped reference, but that any operation on the referend involves proper type checking since the type is given when identifying the field. This provides the mechanism sought in [61] of programs being able to store and pass tokens for objects in the data without needing to have access or know of their internal structure. Such a mechanism is necessary for information hiding and proper partitioning of the design and specification of the collected data. This feature is not a NEPAL innovation but a carryover of one of the powerful features of S-Algol.

With these explicit controls on the use of identifiers, most of the access and control mechanism are achieved, and much of the checking happens during compilation. In particular the fields which are exported can be limited to those which only perform legitimate transformations on the data. In the "lexicon" example the two transactions "add.word" and "add.list.words" can ensure that attempting to add a new word to the vocabulary which is already in the vocabulary has no effect. Any constraints on data may be preserved in this way.

Transactions

Whenever operations occur on data of large volume and long persistence, or on data that is shared, the notion of transaction must be introduced. A transaction provides two functions:

i) It designates a unit of change. All changes in the data within this unit must be recorded or the data must be left unchanged.

When a transaction is started all changes are accumulated. If the end of the transaction is reached then all changes are committed. If the abort statement is encountered at any procedural depth then all changes so accumulated are forgotten and an immediate exit from the transaction takes place.

If the transaction was intended to yield a result then the result is the default value indicated in the transaction parameters pack. See "add.word" in "lexicon" in figure 1.

ii) It establishes some lock to prevent destructive interference between concurrent changes. The mechanism is well known.

Since a programmer should be able to write a transaction not knowing the context in which it should be used it is important to permit transactions to be nested. A nested transaction when committed will be an incremental change in the next outer transaction. The ability to abort individual increments at each level could be particularly useful when writing interactive design programs. It also could be the very devil to implement.

Provision of Views of Data

The archetype and export lists of a class may be used to define a derived class with apparently limited or different fields. For example

```
class sub.lexicon(
      archetype lexicon
      proc sub.find.all( string t,u-> pntr)
        ! this finds all non-4 letter words
         if length(t)=4  then nil  else
        find.all(t,u)
    )

class dictionary(
      archetype sub.lexicon
      proc find.all(string template,using->  pntr)
           sub.find.all(template,using)
     -> find.all)
```

would provide a new view of the "lexicon" class which restricts its users to only obtaining lists of words which match a given template and use only a given collection of letters.

The class may be arbitrarily transformed by the interposition of procedures and transactions. A group composed of a collection of such classes would then provide a defined working context similar to the view mechanism in databases. Access rights to groups must be controlled. Given this, the procedures within the classes can impose any desired privacy constraints.

Accommodating and containing change

With a small collection of data or with temporary data it is acceptable to recover from errors in the original system design or adapt to changing needs by modifying the data description and repopulating the data. With large scale persistent data this is wholly unacceptable. Apart from the computational cost, much of the data will have been derived from sources no longer available, such as people at terminals or from sensors.

In this context it is necessary to be able to make changes without catastrophic consequences rippling through the rest of the system. Change must be localised – e.g. so that code not directly effected need not be changed. Costs must be minimiaed – e.g. in a database describing 300000 race horses (many of which are dead) a change to the data describing horses need not propagate immediately to all horses, since most of them will never be referenced, or will not be referenced before the next change in the data description.

In Nepal, the unit of data description and of program is the class. Arrangements are made to permit a class to evolve to meet its needs. The programming environment of Nepal was to include an integrated editor and compiler. The edits would perform one of the following:

i) Change an existing procedure or transaction
ii) Add a new field (procedure, transaction, variable or constant)
iii) Remove such a field
iv) Change the initialisation arrangements of an existing field
v) Change the list of imported archetypes or of imported types
vi) Change the list of exported fields.

The first two of these have no effect on the existing population of instances of the class changed. The remainder may have an effect. To achieve the necessary effects, the editor/compiler treat the edit session as a transaction. At the end of the transaction a new version of the class is created, it has at present an empty population of instances.

Let us suppose that an editing session added a new field to a class. If a procedure is compiled in the context of this new class definition it may "know" about the new field. Suppose it contains code to dereference this new field. What will happen if the procedure is called with an existing instance of the class, which does not contain the new field?

The run time type checking will determine that the type of the instance was not what was expected, the instance does not contain the desired field. But all is not lost. The language syntax compels the declaration of an initialisation expression for all fields. The run time error handler has only (!) to execute that initialisation expression to get a meaningful default value for the field. It then merely has to extend the structure to include the new field.

All creations will result in latest versions.

An example to illustrate this is given as figure 3. Editing is an essentially dynamic process which cannot be shown as text so we show a text corresponding to the end of the transaction, which progresses "lexicon" from the state it was left in after session 1. (Figure 1)

```
be scrabble
   edit lexicon (
        archetype letter.constants, utilities
        import word
        let vocab =  vector 1..15  of       !index on length
                     vector a..z  of nil  !then on first letter


        proc total(-> int)                  !counts no. of words in vocab
             {. ........}

        let count:=total                   !initialised to number there
                                           !already then incremented by
                                           !insert word

        proc find.word( string s-> pntr)
             {.................}
        proc insert.word( string s)
             {...........
                    abort
              count:=count+1; ...}
        trans add.word (string s -> default false)
              {................................}
        trans add.list.words (string file)
              {................................}
        proc find.all (string template, using-> pntr)
              {................................}
        proc print.words (string query)
              {................................}
       -> add.word, find.all, add.list.words, print.words, count)
quit
```

Session 3 - To Amend the Class Lexicon
Example 7.3

Here a new procedure "total" has been added to lexicon and the variable field count. An initialisation given is applicable to existing vocabularies, so that, should someone refer to

English (lexicon_count)

then the existing instance will be transformed and the words counted, the correct value of count being yielded. The new instance is now associated with the revised class which contains a revised version of insert.word which will maintain this count.

Delayed Evaluation

In order to allow class redefinition the idea was introduced of invoking the initialisation expression for a field if a particular instance of a class did not yet have that field. It was then realised that this mechanism could be generalised. It is never necessary to initialise a field until the first time it is read from. Why not build in delayed field evaluation as a standard feature of all fields, and if it is being applied to fields why not to all identifiers ?

It would obviously complicate the implementation to introduce delayed evaluation if it were not for the fact that it was going to be necessary in some cases anyway. If we made delayed evaluation the general rule, it would have the effect of imposing a data flow order of evaluation [62] rather than a textual order of evaluation on the procedures and class bodies in a database. This idea seems attractive in that it distinguishes the abstract structure of a NEPAL procedure or class body from its textual representation.

This whole edifice of delayed evaluation is based upon the idea that the provision of initialisation expressions is the appropriate way to deal with unknown values. Initialisation expressions with delayed evaluation would certainly be a very powerful way of dealing with unknown values, since an expression could be any arbitrarily complex algorithm. The possibility remains however that the value of the field might be not only unknown but unknowable. What if the field was date of death and the person was still alive ?

There is a strong case to be made out for the introduction of a definite unknown or null value into the language to deal with such instances. Codd has recently proposed a mechanism for dealing with unknown values [63]. The essence of his idea is to add one more member, the unknown member, to the sets of values that may be assumed by any type. This involves modifications to the semantics of the standard operators over the predefined types to handle the case where one or both of their arguments is unknown. However, provided that this is done consistently throughout this extension provides no problem. From an implementation viewpoint, Codd's approach has definite attractions.

Possible problems with Nepal

1. It could be argued that the archetyping mechanism violates the principle of abstract data types that function should be separated from implementation. If the class student has the class person as an archetype then it is possible to treat a student as a person. In which case the implementation of the class student as a specialisation of the class person is revealed. It may be objected that this means that one has no guarantee of the validity of a class as a abstract type, since the exported fields of its archetypes are not protected from modification.

   Whilst this isaserious objection it is not overwhelming. We would argue that Nepal provides adequate mechanisms for the construction of fully protected abstract data types and that semantic hierarchies are a worthwhile additional feature.

There are two distinct means by which full separation of function and implementation can be achieved. The first does not use archetypes. Consider the class dictionary defined above. In this case, the lexicon from which it is created is 'unprotected'. We can easily create a class definition in which the implementation of the dictionary using a lexicon is completely hidden, if we use a pointer.

```
class dictionary(
   import lexicon
   let my.lexicon= default lexicon
   proc find.all( string template,using -> pntr)
     my.lexicon(lexicon _ find.all) (template.using)
  -> find.all)
```

Alternatively using the old definition of dictionary we could use the group mechanism to construct a scope in which the class lexicon was invisible.

```
let English.dictionary = dictionary
English.dictionary(lexicon _ add.list.words) ("Chambers")

group English.space <- dictionary, English.dictionary
```

The group English.space will now have access to the English dictionary but lacking access to the class lexicon it will be unable to coerce the English dictionary to its underlying lexicon.

2. The transitive nature of the archetype relationship poses certain problems. We have defined the classes person and student. If we have the further classes

```
class graduate.student(archetype student
        let first.degree= default "Bsc"
       -> first.degree)

class woman(archetype person
      let children := default vector 1::0 of person
     ->children)

class female.graduate.student(
      archetype graduate.student, woman
    )
```

Here we find that a female graduate student is coercible to a person via two routes: as a woman or as a graduate student and then a student. But the person that we arrive at by these two routes must be the same. In practice this means that the compiler when producing code for the generator for a class must ensure that there are no duplicates created of any of the archetypes. It must therefore evaluate the transitive closure of the archetypes as a lattice, and then recursively generate code for the greatest

upper bounds of all the sub lattices.

It is because of the problem of duplicated archetypes that we do not provide any operator to produce a class from its archetypes. If we provided this we would have to rely purely upon good programming to avoid duplicates.

3. The first criticism of the archetype mechanism is made from the standpoint of the Ada, Mesa school of modular programming (in fact it was made by A. Birrell of Xerox Park). An alternative and I think more telling criticism of the NEPAL scope rules can be made from the standpoint of those in love with classical Algol scope rules. This would be that the whole edifice of NEPAL scope rules is inordinately complex and inhomogeneous, a pastiche of inconsistent and confusing concepts. The resulting syntax is a mass of exceptions. Groups can be declared within groups but nowhere else. Classes can only be declared in groups. Procedures can be declared anywhere as can integers, strings, vectors etc., why not classes ?

In NEPAL two incompatible scope graphs: the Algol nesting tree and the archetype lattice, are superimposed and the resulting conflicts require elaborate rules to resolve. It can be argued, indeed this thesis has argued, that languages should gain their power through simplicity and generality rather than through special features. Since one is starting out from Algol would it not be better to build upon what the Algols are strong in, rather than mixing in another paradigm?

One form of abstraction that Algol does support is functional abstraction. DAPLEX has shown the power that can arise from consistent and orthogonal application of functional abstraction to persistent data. It has been argued, I think convincingly, that by making procedures into first class citizens and consistently applying Algol scope rules, PS-Algol can be extended to achieve most of the effects desired from NEPAL [64]

Suppose we wish in an extended S-Algol to create an abstract type bank.account with the operations create, credit and debit available on it. Crediting an account will only be possible if another account is simultaneously debited, and if the keycode for the debited account is known. The use of procedures as first class citizens allows this to be implemented as follows:

```
structure account.ops( c ( int ->  pntr) create;
                       c ( pntr,   pntr,   int,   int) transfer))

structure account( cpntr account.data)

procedure build.account.package (->  pntr)
begin
   structure acc.data( int keycode, balance)
   procedure New.acc ( int kc)
     account(acc.data(kc,0))

   procedure transfer( pntr creditor, debtor;  int amount, kc)
   begin
      let d.data=debtor(account.data)
      if d.data(keycode)=kc  and d.data(balance)>=amount  do
      begin
         let c.data=creditor(account.data)
        c.data(balance):=c.data(balance)+amount
        d.data(balance):=d.data(balance)-amount
      end
   end
  account.ops(New.acc, transfer)
end
```

This results in a a new abstract type with two operations on it, create and transfer which can not be forged by any other code as no other code knows about the private structure class acc.data.

4. The protection of shared data provided in the language is weak. The basic mechanism is the transaction procedure which locks the context within which it is located. The idea is that a portion of a shared database that is to be protected will have a single root, all actions on it will be encapsulated within transaction procedures declared within the class an instance of which provides the database root. This then becomes not only the root but the only route into the (sub)database. Only one process at a time can then affect the shared data.

   In a sense this mechanism is not so much weak but strong. Its very generality gives it power. With it one can construct any protection graph that one wants. The problem is that it may leave too much to good programming practice. A protected route into a subdatabase is not much good if one of the transaction procedures returns a result that, however indirectly, provides a pointer into the supposedly protected data structure. There is a need here for a simple but safe transaction mechanism. It is also worth noticing that the proposed mechanism provides no way of detecting or preventing deadlocks, though as the next chapter will show one could construct deadlock free protection graphs using it.

Status of Nepal.

Implementation of Nepal has been postponed until more experience has been gained with PS-Algol. An attempt was made to develop an interactive compiler based upon the S-Algol compiler. The modification of this to handle the syntax and scope rules of NEPAL was not difficult. A difficulty came when an attempt was made to incorporate an interactive editing facility into the compiler. It became evident that the basic structure of a batch compiler was ill suited to running as an interactive one so further attempts at the use of force were abandoned. It was decided that the first release of the Nepal compiler would not implement any facilities for editing data or program definitions. This compiler was developed to the stage where it would carry out syntactic and semantic checks on Nepal programs, and translate them into standard S-code. At this point a second difficulty was encountered. It was thought that some extensions to S-code would be needed. In particular it was thought that the dereferencing instructions would have to be modified and it was considered that it would be premature to settle upon a new standard for S-code until more experience had been gained with the implementation of PS-Algol. The Nepal compiler was implemented in S-Algol. For it to be operational it would have to be converted to run as PS-Algol. For both these reasons, priority was given to the development of the PS-Algol system.

The success of the PS-Algol implementation, though it took much longer than anticipated, raises the question of whether an attempt should be made to implement NEPAL. The wisdom of such an attempt is now open to doubt. PS-Algol is not enough, but there are sufficient doubts about the wisdom of the NEPAL scope rules, to justify another attempt at designing an improvement on S-Algol rather than going on immediately with a Nepal implementation.

Chapter 8

Problems of Large Persistent Address Spaces.

Chapter 6 presented an account of the run time environment used to support the persistent heap in PS-Algol. This area is now fairly well understood. The problems that remain are to do with how one manages the larger part of the address space, the address space in its persistent form. We have to ask what sort of addressing topology we will allow in this larger global address space. In the current PS-Algol implementations we deal with only one database at a time, and this database is an exact model of the store assumed by the S-Algol language - with one exception: it is finite. The addressability rules for the store are as defined by the scope rules of the language and the contingent datastructure that may have been constructed on the heap. In fact, given the current implementation of the CMS the space available in a database is limited to one megabyte. Without changing the interface between the CMS and the rest of the system this could be upped to an address space of 8 million objects. That is the maximum number of objects provided within a CMS "bag", given the existing format of a PID. PIDs, however, are capable of transbag addressing, so the number of objects addressable within a bag must be multiplied by the number of simultaneously addressable bags in order to arrive at the total object address space. This total address space is 32 bits split between 8 bag bits and 24 bits to specify the number of objects within a bag.

At present we are only capable of using a fraction of that total address space. What I now wish to examine is how to partition such large address spaces. The problems that have to be dealt with in this context are those of data sharing and concurrency on the one hand, and global garbage collection on the other. It is assumed that programs in languages supporting persistence may be run in a distributed environment.

I would not claim to have done more than investigate some possibilities in this area.

Large Distributed Address Spaces and Concurrency

In what follows I am only dealing with integrated programming/persistence management systems.

Minimal Groups

The basic unit of address space is assumed to be something like a Napal group. It may be assumed to have the following minimum attributes.

1. It may contain definitions of classes or abstract types.
2. It may contain named instances of those types.
3. It may contain unnamed but indirectly accessible instances of these types.

Graphs and scope rules.

I hypothesise that in a persistent programming environment, the fact that the programs and data form part of a homogeneous typed environment enables the use of language features such as scope rules to impose compile time constraints which can reduce concurrency control and garbage collection problems. This should provide an alternative to such methods as the manual definition of transaction classes for conflict graph analysis [65], or the maintainence of global time to serialise transactions [66]

If identifiers or classes in one group are visible within another, then it is possible to represent the visiblity conditions as a directed graph.

In Fig 8.1 the nodes indicate groups G1 and G2. The edge from G1 to G2 indicates that there is some identifier declared in G1 that is visible within G2. In languages supporting a heap (the only ones that are interesting) there need to be additional rules concerned with procedure calling to impose these visiblity rules on the heaps of G1 and G2. Those are gone into later.

The scope/visibility rules of the system define the class of directed graphs onto which visibility conditions subject to these rules can be mapped.

Problems with scope rules

Scope rules for distributed address systems have to address themselves to the following problems.

1. User convenience.
2. Garbage collection.
3. Concurrency control

The garbage collection problem is how to do garbage collection on a large distributed disk based heap in a reasonable amount of time. I assume that a hopeful approach to this would be to allow piecemeal garbage collection of parts of the heap. This implies that there must be some way of partitioning the heap into compartments whose garbage collection can be carried out without prejudice to information held in other compartments. This approach has already been advocated by Bishop in [67] He termed these compartments areas, and proposed that each area have associated with it a table of references made from within the area out to other areas.

One can of course just hope that the garbage collection problem will go away by the time we are in a position to construct large distributed systems, due to advances in storage technology making it impossible or unnecessary to recover space. Let us look at various possible scope rules and their implications.

A rule that says no identifier is visible outside a group, and that only one process may exist at a time in a group, simplifies garbage collection and concurrency control, but lacks user convenience. This is what we have with PS-Algol.

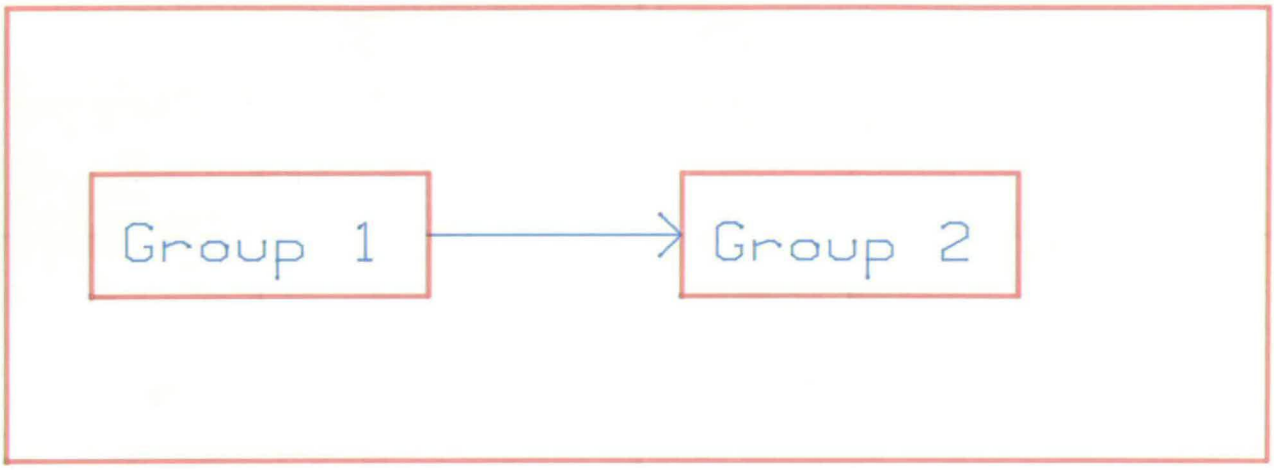

Fig 8.1
Group 2 sees Group 1

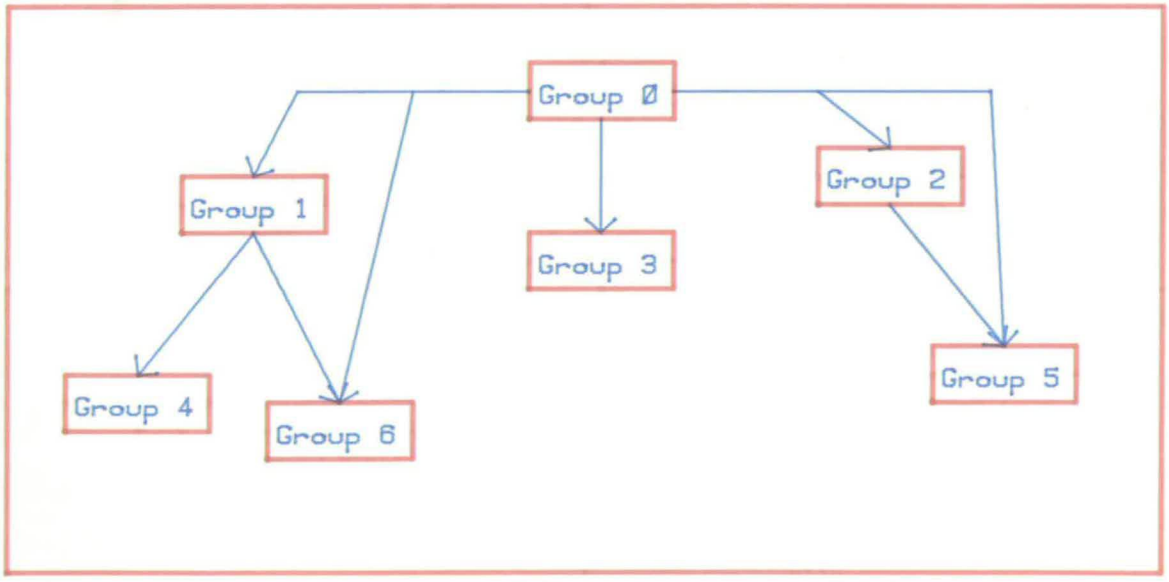

Fig 8.2
Nepal Scope Rule Graph

At the opposite extreme a rule that all identifiers in a group are visible from all other groups and that only one process can exist per group, is powerful but deprives users of the advantages of protection besides being impossible to garbage collect.

From the users point of view it would probably be best if:

1. A user could make identifiers available to selected others.

2. A user could make some identifiers available publicly.

3. Users could have as many processes as they wanted in their address space.

This is a hard set of requirements to meet.

A conservative approach

In the Nepal design we pursued what we thought would be a safe approach. In so doing we imposed a number of restrictions which may be undesirable. The visibility conditions induced by the Nepal scope rules can be exemplified by a graph of the sort shown in Fig 8.2.

This is similar to a tree except that there may be direct links from a root to a leaf bypassing intermediate levels. From the point of view of managing the garbage collection problem this structure has definite advantages. Assume that we try to garbage collect individual groups (since goups can have processes in them I treat them as subjects). When collecting garbage a group need only ask those groups below it if they have any references to items that it holds. For most groups this means that garbage collection can occur without reference to any other groups as most goups are leaf nodes.

From the point of view of concurrency control we still have problems. Let us make the simplifying assumption that for each group there exists some object table that must be locked to carry out a transaction on any item in the group. This pushes the locking granularity up to group level. This would not be appropriate in some applications, but in others, CAD for example, such coarse locking would be adequate.

1. A partial ordering is imposed upon the groups.
2. If we define the root as the UPB under this ordering, then access paths are monotonically increasing under the ordering.
3. The interaction of access paths from processes in different groups is deadlock free.

The Nopal scope rules can thus in a sense be considered safe. It is still necessary to specify the rules that prevent anything in a leaf group's heap being visible from the root groups. What is the problem here?

In essence it arises from the existence of reference or pointer variables in the root group. Consider the following example.

```
group one  begin
   class cons  begin
     let hd:=  default nil
     let tl:=  default nil
   -> hd,tl  end

    let apntr:=cons

    group two <- cons,apntr
    begin
        procedure build.list(->pntr)
      begin
       . .
      . .
       end
       apntr:=build.list
     end
end
```

In this example, a list built in group two is assigned to a variable in group one. Group one ends up with a pointer into group two. How could we avoid this?

We could decide that only constants in group one could be visible in group two, but this wouldn't work. A constant pointer could refer to structures with variable fields. If you had access to the pointer you would have access to the variable fields. So much for that idea.

We could hide all identifiers except for functions from inner groups. The problem with this is that Nepal or S-algol functions can have side effects. A function declared in group one would have access to all the identifiers of that group. It could potentially be called in group two and assign a heap object in group two to a variable in group one. Even if we were to insist that only pure functions without side effects were to be visible, this would still not solve the problem, as a function declared in group one could deliver as its result some object on group one's heap. If this object had variable fields in it we have broken our rules again.

In the end the only solution seems to be to rule that groups are only to have read access to their mother group. With the sort of run time support that is provided by the CMS, this is a trivial rule to enforce. If a process starts in group two, we allow it to do what it likes to group one, but just do not commit the transaction on group one at the end. As a result, the publicly visible version of group one does not change, and can never acquire any pointers into a higher group. If we never allowed daughter groups to alter the state of the mother groups, then there would be no concurrency problem left.

But we can not be this strict. The usefulness of the system would be greatly restricted if there was no write access to data shared between groups. We need to provide a loophole for such updates. The idea of transaction procedures can

be developed into a a suitably safe loophole. A trasaction procedure should have the following properties:

1. It should have the effect of locking the group in which it is declared. In the original NEPAL proposal we only intended it to lock the class instance inside which it was declared.

2. Its parameters would have to be called by value if the transaction was called from another group. By this we mean that if the parameters were pointers then a copy must be taken of the datastructure that they refer to. In fact the copy would have to be of the transitive closure of the parameters on the heap under reachability.

This tightening up of the semantics of transaction procedures, along with making them the only form of write access to ancestor groups makes the Nepal scope rules safe for concurrency and garbage collection.

An Adventurous Approach

The Nepal structure may be safe but it is pretty limiting if you are going to have a distributed environment. For distributed systems it would be difficult to have a tree structure. It would be preferable to have a general network. The question arises: can we impose constraints upon a network so that it models a tree but is not as restrictive as a tree.

I will try to define this environment in DAPLEX.

```
DECLARE Type()=>>ENTITY                                    (1)
DECLARE Basetype()=>Type                                   (2)
DECLARE Components(Type)=>>Type                            (3)
DECLARE Group()=>>ENTITY                                   (4)
DECLARE Owntypes(Group)=>>Type                             (5)
DECLARE Exporttypes(Group)=>>Type                          (6)
DECLARE Importtypes(Group)=>>Type                          (7)
DEFINE Locals(Group)=>>
      UNION OF Owntypes(Group), Basetype                   (8)
DEFINE CONSTRAINT Validexport(Group)=>
      TRANSITIVE OF
        Components(Exporttypes(Group))
       SUBSET
        Locals(Group)                                      (9)
DEFINE CONSTRAINT Validimport(Group)=>
      Importtypes(Group) SUBSET Exporttypes                (10)
DEFINE CONSTRAINT Validowntype(group)=>
      Owntypes(group)
        DISJOINT FROM Importypes(group)                    (11)
DEFINE Components(basetype)=>NULL                          (12)
```

This sets up a set of groups each of which owns a set of types. In addition a group may export or import types. The important thing is the constraints. A group can only import a type that has been exported. A group can only export a

type if that type references only locals or base types. We now set up the definitions for identifiers.

```
DECLARE Id()=>>ENTITY                                          (13)
DECLARE Ownids(Group)=>>Id                                     (14)
DECLARE Typeof(Id)=>Type                                       (15)
DECLARE Exportids(Group)=>> Id                                 (16)
DECLARE Importids(Group)=>>Id                                  (17)
DEFINE CONSTRAINT Validexpid(Group)=>
    Typeof(Exportids(Group)) MEMBER Exporttypes(Group)         (18)
DEFINE CONSTRAINT Validimpid(Group)=>
      Importids(Group) SUBSET Exportids                        (19)
```

These constraints protect us against the dreaded funny loops. What do I mean by funny loops?

Funny loops are the basis of deadlock and garbage collection problems in a network. Lets look at Garbage collection first

In Fig 8.3 we have two groups A and B. A has an object x visible in B. B has an object y visible in A. The two objects x and y refer to each other. Since a distributed garbage collector must work by A asking B "which of my Ids do you reference" and B doing the same to A when B garbage collects, the pair x and y cannot be garbage collected even if no other item points at either of them.

To transform this into a concurrency funny loop we must:

1. Introduce two new groups C, D.
2. Assume that the objects x and y have transaction fields.
3. Assume that transactions operate on the groups in which they occur.
4. Assume that the transaction fields of x attempts to access y and vice versa for the transaction field of y.

If C starts a transaction on x and D starts a transaction on y, then you get deadlock.

The type visibility rules outlined above should outlaw these funny loops. We can show this as follows. In the example above:

x is an export of A and y is an export of B

so from rule 17

```
typeof(x) is a member of exportypes(A)                         (20)
typeof(y) is a member of exporttypes(B)                        (21)
```

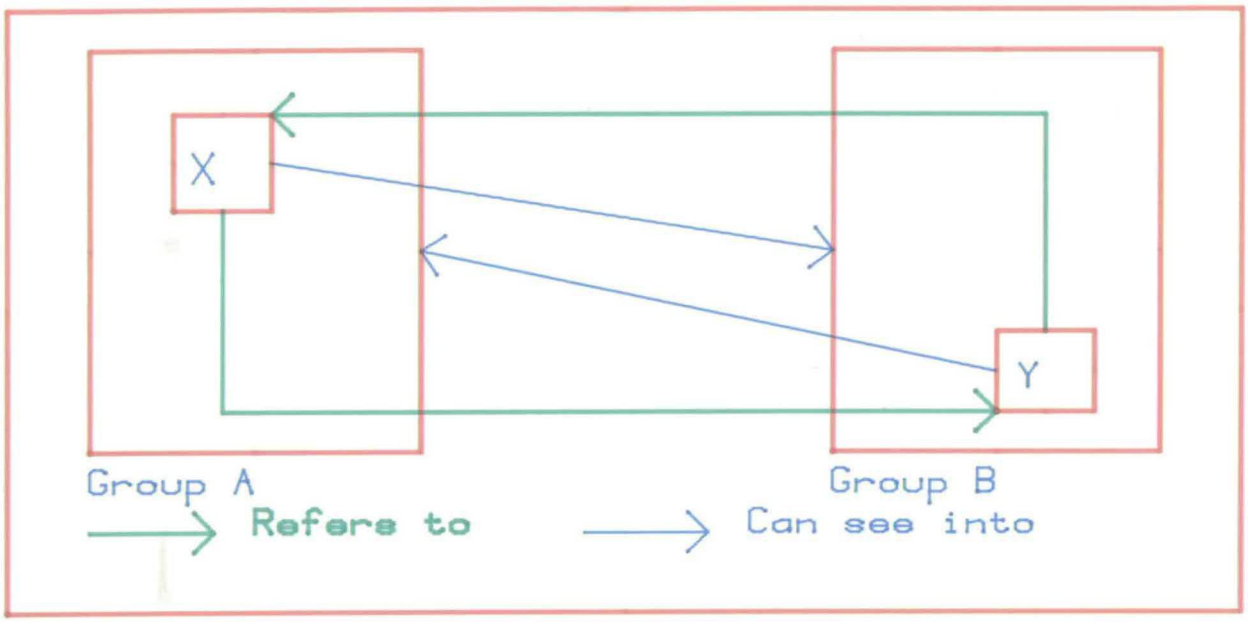

Fig 8.3
Garbage Collection Loop

If x refers to y then assuming a typed language

typeof(y) is a member of components(typeof(x)) (22)

and thus from 20 and 9

typeof(y) is a member of locals(A) (23)

but we already have it that

typeof(y) is a member of locals(B) (24)

from 21 and 9 which leads us to the contradiction that typeof(y) is local to two different groups.

We can conclude that the scope rules specified do protect us against uncollectable or deadlock producing loops. They do have the disadvantage of excluding the use of Groups as a tool for successive abstraction in the provision of library types. One cannot have an abstract type in A that is refined in B and then the refined type used in C.

Conclusion

By working at the level of language design, and employing strongly typed languages, it is possible to impose constraints upon the types of connectivity that can be formed between objects in large partitioned persistent address spaces. Type rules and scope rules, by abstractly defining the range of legal operations in the language, can constrain run time programs to construct only such structures as are incrementally garbage collectable, and deadlock free. The price that is paid for this, as with all strong typing, is a loss of power.

Chapter 9

Results and Prospects

The research project upon which this thesis is based started out with some elevated goals. Experience has shown that these were rather too elevated at some points, but that one of these goals, the most fundamental, is attainable. It was hoped that it would be possible to build a network environment for persistent programming. This was to be based around a database server or chunk manager. This would provide reliable storage of chunks of data sent to it by client machines on a Local Area Network. These client machines would be running programs in Pascal and Algol-68 and their heaps would be looked after by the Data Curator.

This was going to demand a considerable level of intelligence on the part of the Data Curator. It was going to have to know about the typed environments generated by all of the various Pascal and Algol-86 programs on all of the client machines. This in turn implied that the compilers for these and any other languages supported would have to enter into a dialogue with the data curator during the course of compilation, to inform it of what types were defined in the program under compilation. The hope was that by creating this new environment a large body of software, particularly CAD software, written in these languages could with only slight modification be made to run with a persistent heap. It was further hoped that existing compilers for these languages could be used, again with only small modifications to cause them to save their symbol tables in the Data Curator.

As if this were not hubris enough, it was hoped that the data stored in the Data Curator would be accessible to programs in multiple languages on multiple machine architectures. Data generated in an Algol-68 program running on Vax was to be usable by a Pascal program running on an Interdata.

The implementation of the program of research has shown most of these goals to be either too ambitious, not feasible or not worth-while.

1. A Network Environment
   With the advantage of hindsight it looks as if this idea was little more than a kow-towing to the fashions of the day, necessary perhaps to attract funding, but essentially tangential or even orthogonal to the main purpose of investigating persistence in programming languages. As it turned out, lack of equipment in the early stages of the project prevented the Data Curator being connected to a network. Despite this, all the Data Curator software was initially written on the assumption that it was going to work over a network. In the absence of an adequate message passing facility in the operating system, this was found to impose a excessive overhead cost on calling Data Curator facilities. The Vax version of the software abandoned even the pretence that it was designed to work over a network, and was much the better for it. If this experience shows anything, it is that it is unwise to pursue mutually independent research goals within a single program of research.

2. An Intelligent Data Curator

The idea of making the Data Curator intelligent, in the sense that it should know about types, names, environments etc, turned out to be based upon a mistaken idea of the future evolution of computing costs. It was thought that the translation of Persistent Identifiers to Local Object Numbers would always be an expensive task both in terms of memory and cpu cycles. It was assumed that the client machines would have limited cpu power and small address spaces, making it impossible for them to carry out this translation. In fact cpu power and memory are the components of computers whose price is falling most rapidly. Moreover, they are cheaper for small computers than for large ones. The cost of a given amount of memory and processing power in the form of 16 bit micro computers is a fraction of what you have to pay for an equivalent power in the form of 32 bit minicomputers.

What remains expensive is disk storage. It is thus sensible for a Data Curator to have the minimum of intelligence but a high input/output capacity and a large disk capacity. The only Data Curator software that was actually put into use was the untyped Chunk Management System, aimed to meet this requirement.

3. Use of Existing Popular Languages

The two initial target languages of the research were Pascal and Algol-68. These were widely known and had existing user communities. It was hoped that it would be possible to add persistence to the languages with only minimal modifications to their syntax and semantics. Investigation revealed that the changes required were not trivial. It would be necessary to create some form of persistent scope within which the types and roots of the persistent datastructures would be declared. If one made such changes, it would be a touch of sophistry to claim that one still had the same language. A considerable effort would have to be expended to convert programs from the old non-persistent form to the new.

The reason why such changes to the language become necessary are that these languages are strongly typed and type checking is done at compile time. Thus all types used in the persistent heap and all roots of the heap must be explicitly declared at compile time, hence the need for a persistent scope for these declarations. In the language acutually used in the research, S-Algol, there is an element of run time type checking that is just sufficient to allow the provision of a set of generic procedures as the means by which roots into the persistent heap can be established.

We have shown that it is possible to take an Algol and make no change to its syntax yet still provide it with a persistent heap. But it was only possible to do this because we chose a version of Algol with a rather modern and elegant design. The hypothesis that any Algol-like language could be given a persistent heap with only minimal syntactic changes, has not been proved valid. It has not

been proved false, since it is not clear how one can disprove such a hypothesis. An inability to see any way of doing something does not imply that there is no way of doing it. However, on the basis of over a year spent attempting to modify the syntax of Pascal and Algol-68 to support persistence it does seem unlikely that this can be done without such big changes to the syntax and scope rules of the language as to rule out the use of existing compilers.

4. Use of Existing Compilers

The hope that existing compilers could be used has, paradoxically, proved both true and false. It is true that we have been able to use the existing S-Algol compiler unmodified, but this has only been possible because the compiler provides the necessary information for run-time type checking. This was vital for the new garbage collection algorithms and allowed us to postpone until run time, the binding between roots and their symbolic representations. Examination of other compilers showed them to be too monolithic to permit ready modification to handle their symbol tables as a persistent database.

5. Cross Language Portability of Data

This has proved to be infeasible. The differences between the type rules of the languages under investigation prevent the construction of a more general set of type rules under which canonical representations of types may be unambigously compared. In the absence of some method for evaluating type equivalence, no secure cross language portability of typed data seems possible.

6. Orthogonal Persistence

One thing which has been proved is that it is technically feasible to implement persistence as an orthogonal property of data. S-Algol with a persistent heap has been implemented on two machines. Addressing mechanisms and garbage collection algorithms have been developed which support transparent access to and secure storage of persistent data-structures. These implementations are constantly being improved as we gain a better understanding of the practicalities of supporting persistent heaps. Moreover, the subjective experience of those who have worked with PS-Algol is that it is much easier to write programs that work on persistent data in PS-Algol than in Pascal.

It can therefore be concluded that the most fundamental hypothesis of the research program has been justified.

It is possible to construct an Algol with a transparently persistent heap, and that such a language is of help in the development of programs operating on persistent data.

7. Longer Term Persistence

PS-Algol provides no means of modifying datastructure definitions whilst still being able to preserve persistent data, nor does it provide a structured enough access to the persistent data. In the long term, any software system using persistent data will require some

means of modifying its data definitions. Any serious multi user system will require a controlled and partitioned access to shared data and the provision of views of data. These features cannot be provided by minor extensions to some existing language. They necessitate the development of a language supporting the same set of features as NEPAL, though these need not be provided in the same way. PS-Algol should provide a base upon which such a language can be developed.

Prospects

In my first chapters I argued that a large part of what is euphemistically called computing, is still tied up with the processing of serial data files. The greater part of data processing is still in the age of the Turing machine. Whether it is a matter of commercial applications, editing or compiling, it is the persistent store, the disks not the RAM that holds the significant data. So long as this is represented within programming languages as serial data streams, or at best untyped random access store, the main body of computing cannot be said to have entered the Von Neumann age. The Von Neumann model of computing has so far only been applied to volatile store. Around this there has grown up a whole linguistic culture to provide sophisticated abstract representations of this store. The development of orthogonal persistence liberates the full potential both of the Von Neumann model and its linguistic culture, and puts this at the disposal of applications programmers.

The smashing of the Turing bottleneck, making persistence trivially available should produce a greatly enriched computing culture. We can foresee the demise of files, and of the whole body of computing techniques that are predicated upon them. Ask yourself for instance: how much of language design and compiler technology is based upon the assumption that the definitive form of a program is a serial file?

How much of the much vaunted Unix culture is just a collection of programs that do things to text files?

To make revolution it is first necessary to create revolutionary public opinion [68]. In the context of computing we must make persistent programming available cheaply and efficiently on widely used brands of micros. In doing this we can create among the broad mass of new computer users, a culture and climate of opinion that takes persistence for granted. Win the peasants and the citadels must fall.

References

[1] Wirth, N. and Hoare, C.A.R. A contribution to the development of Algol. Comm. ACM 9,6 (June 1966),413-431.

[2] Wijngaarden, A. van, et al, Revised Report on Algorithmic Language Algol 68. Supplement to ALGOL BULLETIN 36, (March 1974)

[3] Wirth, N. The Programming Language Pascal. Acta Informatica 1, (1971), 35-62

[4] Birtwistle, G., Dahl, O.J., Myrhaug, B., Nygaard, K., Simula BEGIN, Auerbach Publishers, Philadelpia (1973)

[5] Mitchell, J.G., Maybury, W., Sweet, R. Mesa Language Manual. Xerox PARC, CSL-78-1, (1978)

[6] Liskov, B., Moss, E., Schaffert, C., Scheifler, R., Snyder, A., CLU Reference Manual, Massachsetts Institute of Technology Laboratory for Computer Science, (Jul 1978)

[7] Lampson, B.W., Horning, J., London, R.L., Mitchell, J.G., Popek, G.L., Report on the Programming Language EUCLID, ACM SIGPLAN notices, 12(2), (Feb 77)

[8] United States Department of Defence, Reference Manual for the Ada Programming Language, July 1980

[9] Hoagland, A.S. Storage Technology: Capabilities and Limitations, Computer, Vol 12, No 5, (May 1979) pages 12-19

[10] Whitfield, H., Wight, A.S., EMAS The Edinburgh Multi-Access System, The Computer Journal, Vol 16, No 4, (Nov 1973)

[11] COSDASYL Programming Language Committee, Database Task Group Report, ACM (Apr 1971)

[12] Codd, E.F. A Relational Model For Large Shared Data Banks, Communications of the ACM 13, No. 6 (June 1970) 377-87

[13] Schmidt, J.W., Some High Level Language Constructs for Data of Type Relation, ACM Transactions on Database Systems 2, No. 3 (September 1977) 247-281

[14] Wasserman, A.L., Sheretz, D.D., Kersten, M.L., Reit, R.P. van de, Dippe, M.D., Revised Report on the Programming Language PLAIN, ACM SIGPLAN notices (May 1981)

[15] Shopiro, J.E., THESEUS - A programming language for Relational Databases, ACM Transactions on Database Systems 4, No. 4 (Dec 1979)

[16] Merrett T.H., Aldat - Augmenting the Relational Algebra for programmers,

Technical Report SOCS-78.1, School of Computer Science, McGill University, Montreal (Mar 1979)

[17] Myolopolous, J., Bernstein, P.A., Wong, H.K.T., A Language Facility for Designing Database-Intensive Applications, ACM Transactions on Database Systems 5, No. 2 (June 1980)

[18] Hammer, M.M., Berkowitz, B., DIAL: A Programming Language for Data Intensive Applications, ACM SIGMOD International Conference on the Management of Data, (1980)

[19] Smith, J. M., Fox, S., Landers, T., Reference Manual for ADAPLEX, Computer Corporation of America (Jan 1981)

[20] Hoare, C.A.R., 1980 Turing award lecture: The Emperors Old Clothes, Communications of the ACM, Volume 24 No 2, (1981)

[21] Shipman, D. W., The Functional Data Model and the Data Language DAPLEX, ACM Transactions on Database Systems 6, No.1 (Mar 1981)

[22] Buneman, P., Frankel R. E., FQL - A Functional Query Language, Proceedings ACM SIGMOD Conference on Management of Data (1979)

[23] Smith, J. M., and others, MULTIBASE— Technical Report on Basic Architecture, Computer Corporation of America, (Nov 1980)

[24] Atkinson, M. P., Database Systems and Programing Languages, Proceedings of the 4th Very Large Database Conference, (Sept 1978), pp. 408-419

[25] Cockshott, W. P., Segments, Data Curator Documentation, Year 1, Memo1, (Dec 1979)

[26] Cockshott, W. P., Modular Structure Revisited, Data Curator Documentation, Year 1, Memo 8, (Dec 1979)

[27] Cockshott, W. P., Modular Structure, Data Curator Documentation, Year 1, Memo 15, (Feb 1980)

[28] Ingalls, D. H., The Smalltalk-76 Programming System Design and Implementation, Fifth Annual ACM Symposium on Principles of Programming Languages, (Jan 1978)

[29] Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Gray, J. N., King, W. F., Lindsay, B. G., Lorie, R. A., Mehl, J. W., Price, T. G., Putzolu, G. R, Schkolnick, M., Selinger, P. P., Slutz, D. R., Strong, H. R., Tiberio, P., Traiger, I. L., Wade, B. W., Yost, R. A., System R: A Relational Data Base Management System, Computer, (May 1979), pp. 42-48

[30] Kaehler, E., Virtual Memory for an Object-Oriented Language, Byte, (Aug 1981), pp. 378-387

[31] Albano, A., Occhiuto, M. E., Orsini, R., A Uniform Management of Temporary and Persistent Complex Data in High-Level Languages, in Infotech State of the Art Report series 9 number 8, DATABASE, Pergammon Infotech Limited, Maidenhead, Berkshire, England. (Jan 1982)

[32] Gordon, M., Milner, R., Wadsworth, C., Edinburgh LCF, Lecture notes in computer science vol 78, Springer Verlag (1980)

[33] Althusser, L., Lire le Capital, Maspero, Paris (1965)

[34] Cockshott, W. P., Commands and Services of Curator and Client, Data Curator Documentation, Year 1, Memo 7, (Dec 79)

[35] Cockshott, W. P., Ideas on Type Algebra, Data Curator Documentation, Memo 18, (June 1980)

[36] Monahan, B., A Type Algebra Semantics, Data Curator Documentation, Year 1, Memo 32, (Sept 1980)

[37] Cockshott, W. P., An Example of the Application of The Type Algebra, Data Curator Documentation, Memo 22, (June 1980)

[38] Cockshott, W. P., Use of the Type Algebra to Describe Algol68 Modes, Data Curator Documentation, Memo 24, (June 1980)

[39] Welsh, J., McKeag, M., Structured System Programing, Prentice Hall, London, (1980)

[40] Cockshott, W. P., Indicant Manager Input Syntax and Data Structures, Data Curator Documentation, Year 1, Memo 25, (Aug 1980)

[41] Cockshott, W. P., New Syntax for the Type Algebra, Data Curator Documentation, Year 1, Memo 47, (Sept 1980)

[42] Wirth, N., Pascal Report, Springer Verlag, New York (1975)

[43] Bourne, S. R., Birrell, A. D., Walker, I., Algol68C Reference Manual, University of Cambridge Computer Laboratory, (1974)

[44] Robertson, P. S., The IMP-77 Language, Internal Report, University of Edinburgh Department of Computer Science, (Nov 1977)

[45] Morrison, R., S-Algol: Reference Manual, Department of Computational Science, University of St Andrews, (1980)

[46] Atkinson, M. P., Chisholm, K. J., Cockshott, W. P., The New Edinburgh Persistent Algorithmic Language, Infotech State of the Art Report: Database, Series 9, Number 8, Pergamon Infotech, Maidenhead England, (1981)

[47] Atkinson, M. P., Chisholm, K. J., Cockshott, W. P., PS-Algol: an Algol with a Persistent Heap, Sigplan Notices, (May 1982)

[48] Robertson, P. S., The IMP-77 Language, Internal Report, University of Edinburgh, Department of Computer Science, (Nov 1977)

[49] Challis, M. P., Database Consistency and Integerity in a Multiuser Environment. In: Databases: Improving usability and responsiveness. New York: Academic Press, (1978), 245-70

[50] Marshall, R. M., Respecification of the Chunk Manager User Interface, Data Curator Documentation, Year 2, (Aug 1981)

[51] Cockshott, W. P., The Message Switcher Command Interpreter, Data Curator Documentation Year 1, Memo 16, (March 1980)

[52] Cockshott, W. P., Switcher Internal Architecture, Data Curator Documentation, Year 1, Memo 20, (May 1980)

[53] Cockshott, W. P., Switcher Interface Routines, Data Curator Documentation, Year 1, Memo 23, (June 1980)

[54] Cockshott, W. P., Vax Ether Interface, Data Curator Documentation, Year 1, Memo 52, (Sept 1980)

[55] Marshall, R. R., The CMS Command Interpreter Program, Data Curator Documentation, Year 2, Memo 18, (Aug 1981)

[56] Atkinson, M. P., Chisholm, K. J., Cockshott, W. P., The Chunk Manager, Edinburgh University Department of Computer Science Internal Report, (1982)

[57] Nelson, B. J., Remote Procedure Call, Xerox Palo Alto (May 1981)

[58] Rashid, R. F., Robertson, G. G., Accent: A communications oriented network operating system kernel, Department of Computer Science, Carnegie Mellon University, (April 1981)

[59] Robertson, P. S., Whitfield, C., Mouses User Manual, Moray House College, Edinburgh, (1979)

[60] Owoso, G. O., Persistent Type Checking In PS-Algol, Data Curator Documentation, Year Three, Memo 9, Version 2, (March 1982)

[61] Atkinson, M. P., Jordan, M. J., Martin, M. J., A Uniform Approach to Data and Programs, Dept of Computer Science Internal Report, University of Edinburgh, (1980)

[62] Ackerman, W. B., Data Flow Languages, Computer, Vol 15, No 2, pp 15-26, (Feb 1982)

[63] Codd, E. F., Extending the database relational model to capture more meaning, ACM Transactions on Database Systems 4, No 4, (Dec 1979)

[64] Morrison, R., Private communication

[65] Rothnie, J. B., et al., Introduction to a System for Distributed Databases (SDD-1), ACM Transactions on Database Systems, Vol 5, No. 1, (Mar 1980), Pages 1-17

[66] Kung, H. T., Robinson, J. T., On optimistic Methods for Concurrency Control, ACM Transactions on Database Systems, Vol 6, No 2, June 1981

[67] Bishop, P. B., Computer Systems with a Very Large Address Space and Garbage Collection, Phd Thesis, MIT, 1977

[68] Tung, M. T., Selected Quotations, Foreign Languages Press, Beijing, (1968)