P-PASCAL:
A DATA-ORIENTED PERSISTENT PROGRAMMING LANGUAGE

Sonia Berman

Thesis Presented for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

UNIVERSITY OF CAPE TOWN

August 1991

DST 004 BERM

93/5324

ABSTRACT

Persistence is measured by the length of time an object is retained and is usable in a system. Persistent languages extend general purpose languages by providing the full range of persistence for data of any type. Moreover, data which remains on disk after program termination, is manipulated in the same way as transient data. As these languages are based on general purpose programming languages, they tend to be program-centred rather than data-centred. This thesis investigates the inclusion of data-oriented features in a persistent programming language.

P-Pascal, a Persistent Pascal, has been designed and implemented to develop techniques for data clustering, metadata maintenance, security enforcement and bulk data management. It introduces type completeness to Pascal and in particular shows how a type-complete set constructor can be provided. This type is shown to be a practical and versatile mechanism for handling bulk data collections in a persistent environment. Relational algebra operators are provided and the automatic optimisation of set expressions is performed by the compiler and the runtime system.

The P-Pascal Abstract Machine incorporates two complementary data placement strategies, automatic updating of type information, and metadata query facilities. The protection of data types, primary (named) objects and their individual components is supported. The challenges and opportunities presented by the persistent store organisation are discussed, and techniques for efficiently exploiting these properties are proposed. We also describe the effects on a data-oriented system of treating persistent and transient data alike, so that they cannot be distinguished statically.

We conclude that object clustering, metadata maintenance and security enforcement can and should be incorporated in persistent programming languages. The provision of a built-in, type-complete bulk data constructor and its non-procedural operators is demonstrated. We argue that this approach is preferable to engineering such objects on top of a language, because of greater ease of use and considerable opportunity for automatic optimisation. The existence of such a type does not preclude programmers from constructing their own bulk objects using other types - this is but one advantage of a persistent language over a database system.

ACKNOWLEDGEMENTS

I would like to thank the following people, listed alphabetically, who helped towards this thesis:

Prof Malcolm Atkinson, for inventing persistent languages and for suggesting my thesis topic

Prof Ken MacGregor, my supervisor, for his advice and encouragement

Prof Ron Morrison and his team, for their hospitality and their helpful discussions of my work.

CONTENTS

CHAPTER 1 INTRODUCTION

1.1 PERSISTENT LANGUAGES

Traditionally, programs have made use of a file system or Database Management System for manipulating long-term data. This involves working with two different data models, and requires code for translating between the two views and for transferring data between memory and disk (typically 30 percent of a program [Atki86]). The structure and protection afforded by the programming language types are lost when the data is written to disk. Persistent programming languages solve this impedance mismatch by presenting a single type system and associated operators for both transient and long-term data alike. Experience with PS-Algol [Atki83], a persistent S-Algol[Cole], showed that productivity increased, program code was shortened and maintenance simplified, compared with conventional database applications [Atki83].

Persistence is defined as the length of time a data object exists in a system and is usable [Atki84]. Persistent languages are based on the following principles [Atki83]:

- *Persistence Independence*: The persistence of an object is independent of how it is manipulated. Conversely, a program fragment is expressed independently of the persistence of the data it uses.
- *Persistent Data Type Orthogonality*: In line with the principle of data type completeness, all data should be allowed the full range of persistence.
- *Persistence Identification*: The identification of values which are to persist is not related to the type system, computational model or control structures of the language.

A Database Programming Language brings together ideas from programming languages and database systems and integrates them in a consistent way. The programming language features they build on include computational completeness, strong typing and modularisation mechanisms. Modern languages provide encapsulation, polymorphism and abstract data types. Database systems provide logical data independence and methods of enforcing data security and integrity. They have mechanisms for non-procedural data processing and automatic optimisation, as well as semantic data model notions such as classification, aggregation and generalisation [Smit77]. The most well known data models are SDM [Hamm], Binary Models [Abri], Entity-Relationship Models [Chen] and their derivatives (eg. IFO[Abit] and ESDM[King]).

Persistent languages are a particular kind of Database Programming Language. They introduce a minimum of new constructs into a conventional language, allowing any object to persist without affecting the way it is manipulated. As such, they are easy to learn and use. Another consequence is that these languages tend to be program-centred rather than data-centred [Dear89].

1.2 THE P-PASCAL CONTRIBUTION

This thesis presents research into the inclusion of data-oriented features in a persistent environment. Although persistent languages may be easy to use and their implementations efficient, their applicability will remain limited so long as the facilities offered by conventional database systems are not provided. To support systems with large homogeneous collections of data requires data types and operators different from those of general-purpose programming languages. When confidential information exists, security enforcement is essential. A database that integrates a variety of interrelated data for many different users requires a facility for metadata querying, so that programmers can easily obtain accurate information on the types in use. This thesis develops techniques for the inclusion of a type-complete bulk data constructor, an automatic metadata maintenance system, security enforcement and data clustering strategies in a persistent environment. P-Pascal has been designed to show how these features can be incorporated in a persistent language, and a multi-layered system has been constructed to demonstrate its implementation.

P-Pascal extends the CPOMS persistence mechanism [Brow85] employed by PS-Algol to incorporate the data-oriented facilities discussed above. At the same time, it shows how this mechanism can accommodate a language different from that for which it was designed, eg. as regards data representation and assignment semantics. The need to investigate this capability was noted by the PS-Algol developers in [Atki84]. The majority of persistent languages that have appeared since then developed independent storage management systems [eg. Saje, Ever, Rich89a, Riem, Rose89, Wile]; exceptions are Persistent-Prolog [Gray], and the functional languages Hope+ [Perr] and Staple[Davi90]. During the P-Pascal development, the Napier Stable Store [Brow89] was implemented. It will be shown how the ideas presented in this thesis can also be applied in that system.

1.2.1 THE LANGUAGE

P-Pascal introduces type completeness and orthogonal persistence into Pascal. Pascal was chosen as a basis for the new language because of its simplicity and popularity, and because its set type provides a natural way of introducing bulk data objects. The implementation employs a layered approach, in which successive layers deal with progressively richer data types. This permits modular, incremental design and facilitates experimentation with both the language and its implementation. The layered architecture is shown in figure 1.1. The Semantics Layer has not yet been constructed.

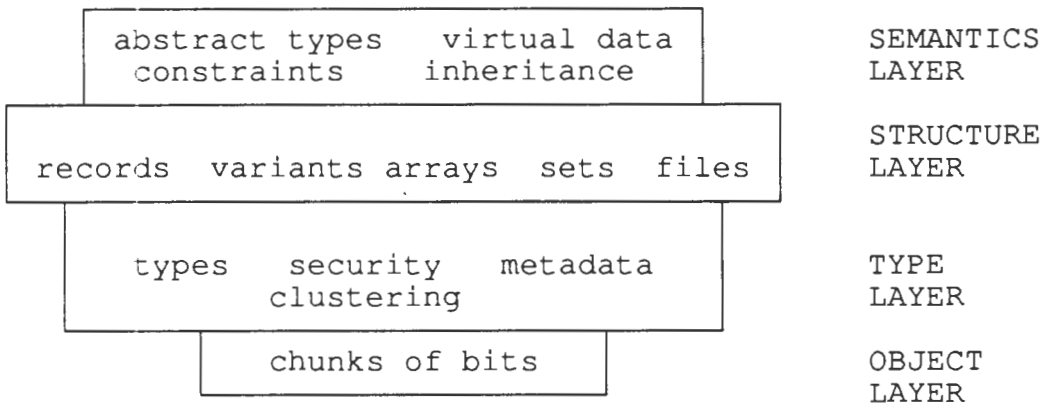


Figure 1.1 The four layers of the P-Pascal architecture.

The P-Pascal set, which can comprise elements of any type, is proposed as a bulk data type for persistent stores. This single type constructor can be used to represent a wide range of data collections of differing characteristics. High-level operators which permit automatically optimised, non-procedural manipulation of data collections are introduced to improve programmer productivity and application performance. The desirability of such a type has been mentioned in many papers on persistence [eg. Atki87a, Atki89].

P-Pascal sets can contain data of any type, including pointers, arrays, sets and variants. Any number of keys, or unique identifiers, can be defined. The declaration syntax is shown below, with optional parts enclosed in square brackets.

```
TYPE setname = [ PACKED ] SET OF basetype
                    [ KEY keylist END ];
```

P-Pascal permits both iterative and non-procedural set manipulation. In addition to the set operators of Standard Pascal (intersection, union, difference and the relational operators), Cartesian product, desetting, ordering and subset selection are introduced. A projection operator for records (and arrays or sets of records) exists, as well as a natural join operator. Some examples of P-Pascal statements are shown below.

```
LEGAL := (S1 * S2) < (S3 + S4);   (* Standard Pascal operators *)

X := (Deliveries JOIN Parts) -> (sno,pno,cost);   (* project from join *)

ALL_SP := (Supps # Parts) -> (sno,pno,city,0);  (* project from product *)

S := S -> (sno, sname, city, discount + 5);  (* update *)

FOREACH S^ [city = 'LA' ] DO       (* iteration *)
        transport := 0;            (* subset updated *)

WRITELN (Suppliers^ ORDER ASC city );  (* print in city order *)

S := (Emps # (Emps AS Mgrs)) [ Emps.mgrno = Mgrs.eno ];   (* data on
        employees and their managers *)

Arrset1 := Arrset2 [ Arrset2 [1] < N ];  (* selection from a set of arrays *)
```

1.2.2 CLUSTERING

A persistent environment with indirect addressing, garbage collection and inter-linked databases presents both some challenges and some opportunites for data placement. Two strategies are proposed in this thesis, for clustering data according to its expected usage. Type-clustering stores objects according to their type; family-clustering aims at keeping data close to the objects it references. These strategies, which can be used independently or applied together, are transparent to programmers.

1.2.3 METADATA

In a non-trivial database environment, the management of names can become problematic. Due to the difficulty of working with a large name space, a system to query type information is needed for persistent environments [Atki87a, Morr87]. The P-Pascal system maintains metadata automatically, and enables the structure and organisation of the persistent store to be queried. The information is represented as a number of inter-related sets. Type names and definitions can be accessed using either set operators or standard metadata query functions. Through the Metadata Query Utility, programmers can learn about a database, recall type specifications, check if a name is in use, etc. The protection, ownership, population, definition and existence of types can be determined, as well as cross-reference data. This has been achieved without introducing a schema concept, which is contrary to the principles of persistence independence and persistence identification. By avoiding schemas the binding of program to database can be flexible, and it is possible to add new types to a database when it is already in use.

1.2.4 SECURITY

In addition to the protection afforded by strong type-checking, illegal access to sensitive data should be prevented. At present, persistent languages do not incorporate authorisation primitives. This research examines the effects of persistence independence on a security subsystem and proposes techniques to meet this challenge. The implementation provides for the protection of databases, types, and "primary" (named) objects. Structure components can be protected individually, and their use and that of the structure itself curtailed accordingly.

1.2.5 BULK DATA

For persistent programming languages to serve as a viable alternative to conventional database systems, and not be restricted to certain kinds of applications, some constructs are required to facilitate the processing of large homogeneous data collections. Questions regarding the nature of this type constructor have appeared in several papers; some examples are given below.

".. identify a bulk data constructor, such as relation or index, define the operations on these and include the constructor in the set of modelling facilities in the language. This

was our first approach [Atki85] but so far has proved difficult to integrate with the Principle of Data Type Completeness." [Morr89b].

"Bulk types ... Questions that then arise, are:

(i) should we attempt to meet these needs with one or several type constructors?

(ii) should we adapt our previous types to have similar operations?

(iii) which bulk type constructors should be built into a language?" [Atki89].

"It remains an open question as to whether they (indexing constructs) should have a separate type or whether both types, relations and indexes, can be subsumed in one generic type." [Atki85].

"In fact the question of whether the bulk type should have an ordering, and whether cursors are even necessary has yet to be resolved. These, as well as problems of implementation, are research issues." [Atki87a]

"We therefore view Napier88 as a shell language on which other experiments such as bulk data constructors for database programming may be constructed." [Dear89].

This thesis aims to answer these questions. The provision of a type complete set constructor in P-Pascal shows how expression optimisation and effective use of the persistent store and local heap can be achieved [Berm89]. The physical storage strategy reduces disk fetches and permits direct access to elements in an environment where other objects are indirectly addressed. A P-Pascal set is stored internally using one of five alternative representations according to the characteristics of the elements and keys, but this is transparent to programmers. Sets can be used for homogeneous and non-homogeneous data collections, as an index type, or to represent relationships between objects.

During compilation, set expressions are first processed by the Translator to produce a canonical form, then an execution strategy is chosen by the Optimiser. These aim to reduce operand sizes as early as possible, to make the maximum possible use of indexes, to represent intermediate results as compactly as possible and to perform consecutive operations at the same time wherever possible. The implementation of sets appears sufficiently general to be incorporated in other languages with a richer collection of data types. An initial version of P-Pascal has been built, and although some functions have

been omitted, the significant aspects of the system have been implemented to illustrate the methods and techniques proposed and demonstrate their viability.

1.3 THESIS OVERVIEW

Chapter 2 introduces the P-Pascal language, focussing on its treatment of persistence and type complete sets. The next chapter presents examples of other persistent programming languages and database programming languages, and compares these with P-Pascal. Chapter 4 outlines the functioning of the P-Pascal interpreter. It describes the CPOMS and shows how this persistence mechanism can be used by a system with complex nested structures.

Chapter 5 describes a data clustering algorithm and shows how it can be implemented. The consequences of garbage compaction and of the physical organisation of the persistent store are significant, but it is possible to implement a reasonable data placement strategy, based on notions such as type-clustering and family-clustering. The former reserves disk blocks for objects of the same type, the latter keeps space on every block for objects referenced by its occupants.

Chapters 6 and 7 present the metadata management and security subsystems. We describe their effect on database retrievals, on changes to the persistent store and on local heap manipulations, and explain when and how type-checking and security enforcement is performed. There is no schema restricting the data allowed on a database, and new types are not introduced explicitly. The metadata subsystem shows the need for concurrency in a persistent environment, and the impact of security information on the processing of types. The effects of bulk data types, complex objects, persistent store organisation and other features such as name equivalence on these subsystems are discussed.

In chapter 8, internal set representation is outlined and system components dealing with efficient evaluation strategies are presented. Some advantages over relational database systems are noted and the application of these ideas in other persistent languages is discussed. The thesis concludes with a summary of the work and some suggestions for future research.

CHAPTER 2 THE P-PASCAL LANGUAGE

This chapter describes P-Pascal, showing how the Persistent Programming Language requirements laid down in [Atki83, Atki85] guided its design. The extensions introduced to facilitate bulk data processing and persistence specification are presented.

2.1 DESIGN GOALS

P-Pascal was developed to introduce orthogonal persistence, persistence independence and data type completeness in Standard Pascal, while adding a minimum of new constructs to the language. Type completeness means that language restrictions based on data type are minimised. The current version of P-Pascal does not consider procedures or functions as first class objects, and hence these cannot be read or printed, declared as structure components or returned as function results. All assignable types are first class objects; a type is assignable if it is neither a file type nor a structure type with a component that is not assignable [Jens]. The type restrictions in ISO Standard Pascal [Jens] which have been relaxed in P-Pascal are:

- values of any assignable type can be constructed since record and array aggregates are supported
- constants of any assignable type can be defined
- records, sets and arrays can appear in READ(LN) and WRITE(LN) statements
- parameters and function results can be of any assignable data type
- equality and inequality are defined for all types
- a variant part may appear in any position in a record declaration
- the base type of a set can be any assignable data type.

The extension of set types to cover elements of any kind provides a natural means of incorporating bulk data into P-Pascal, without introducing new data types. High-level operators for sets have been added to the language, to permit non-procedural manipulation.

As the provision of persistence requires some control over data usage, security enforcement is incorporated. Databases can optionally be "clustered" so that objects are stored near those with which they are likely to be accessed. Metadata is automatically maintained and may be accessed in the same way as data, or using standard metadata query functions, or through a Metadata Query Utility. These language features are discussed in chapters 5 through 7.

The principles of orthogonal persistence and persistence independence guided the design of the language. Orthogonal persistence is defined as the ability to make objects of any type persist [Atki83]. The resulting simplicity and versatility permits an easy transition from Standard Pascal to P-Pascal, making databases easy to use. Persistence independence enables the language to treat permanent data in exactly the same way as it does transient data. That is, the conventional Pascal data types and operators can also be used to manipulate disk-resident data. The remainder of this chapter describes the P-Pascal language. For further details, the reader is referred to the P-Pascal Report in Appendix A.

2.2 DATA TYPES

The data types are exactly those of Standard Pascal. Thus the simple types are integers, chars, booleans, reals, enumerated types, subrange types, and (typed) pointers. The structured types are records, variant records, arrays, files and sets. The Standard Pascal type system is extended to allow any assignable type in any context.

2.2.1 COMPATIBILITY

Type compatibility rules are those of Standard Pascal, for upward compatibility, with an additional rule to cover derived types. A derived type results from applications of the operators project, join, cartesian product and aggregation.

Two types are compatible if any of the following holds:

(1) They are the same type (i.e. the same type name).

(2) Both are string types with the same number of components.

(3) One is a subrange of the other, or both are subranges of the same type.

(4) Both are Standard sets, their base types are compatible and either both are packed or neither is. A Standard set is a set of ordinals, i.e. a set which is permissable in Standard Pascal.

(5) Either is a derived type and the types are structurally equivalent.

Two types are structurally equivalent if they apply the same type constructor, have the same number of component types and the component types are compatible (compared pairwise in order). The mixture of name and structural equivalence arises from Standard Pascal Type Compatibility Rules [Jens] and the addition of Rule 5. The alternative to

Rule 5 is to force all subexpressions involving project, join or product to be prefixed by their type. This makes the syntax cumbersome, requires type definitions for all intermediate results and furthermore does not preclude a structural check, since the type prefix must be validated structurally. The typing of aggregates is also inconsistent with Pascal's notation for set constructors, viz. [e_1, ..., e_N], which is untyped. The alternative of using structural equivalence throughout is contrary to Standard Pascal, and generally less efficient, although the same performance as name equivalence can be achieved under certain conditions [Conn90].

A value of type T2 can be assigned to a variable of type T1 (i.e. T2 is assignment-compatible with T1) if one of the following holds:

(1) T1 and T2 are the same assignable type.
(2) T1 and T2 are compatible string types.
(3) T1 is Real and T2 is Integer.
(4) T1 and T2 are compatible Standard set types or compatible ordinal types and the value is a member of the set of values determined by T1.
(5) T2 is a derived type and T1 is compatible with T2.

2.2.2 P-PASCAL SETS

A set declaration has the form

TYPE setname = PACKED SET OF basetype KEY keylist END;

where PACKED, and the KEY ... END section, are optional.
The base type can be any assignable type, so sets of pointers, sets of sets of arrays of records, etc. are all valid. Unlike arrays, the size of a set is not fixed, its elements are not ordered and a set never contains duplicate elements.

A keylist comprises any number of key specifications, each of which can contain any number of components. This specifies an integrity constraint for the set type: within any one occurrence of this type, no two elements have identical key values. The order of key components, and of the keys in the keylist, is not significant; and one key can be a superkey of another. Any value in an element can participate in a key, including components of components (to any level). Keys are optional, to cater for sets which are small, transient, highly volatile or have only entire elements as keys. A key cannot contain variant fields, since its value must be defined for all elements. The present

implementation supports key components of any assignable type besides sets, for reasons of efficiency.

Examples:

```
dayset    = SET OF daypointer;
mthset    = SET OF Jun..Nov;
moneyset  = SET OF REAL;
payments  = SET OF moneyset;
class     = SET OF stu_record KEY stu_name, stu_initials;
                                                  stu_number
                    END; (* keys *)
staff     = SET OF employee_record; (* no keys *)
faculty   = SET OF dept_array KEY [1] END; (* one key *)
shipments = SET OF shipment_rec KEY deliverynum;
                                                    client.branch, date
                                                  END; (* note nested field in key  *)
```

2.3 AGGREGATES

In addition to the Pascal notation for literals of type integer, real, boolean, char, subrange, set and enumerated type, array and record aggregates are introduced. An array aggregate is a sequence of values separated by commas and enclosed by the symbols [. and .]; a record aggregate is a sequence of values separated by commas and enclosed by the symbols [< and >]. The order of the values in an aggregate is significant, and array elements are given in ascending order of index value (row major order). Aggregates can be nested to any depth.

2.4 OPERATORS

The operators of Standard Pascal have all been retained, and extended where necessary to apply to data of any type. Sets can be used in all the standard contexts: on the left hand side of assignments, in set expressions, as function results and as variable or value parameters. New set and record operators have been introduced, and set operators can be used in any combination, without any restrictions on expression complexity. The programmer should not rely on the order of operator evaluation within an expression as

the compiler can optimise. Assignment has a copy semantics throughout the language, since a pointer type exists and can be used for efficiency as appropriate.

Additional set operators have been introduced to permit higher-level, non-procedural set manipulation. With the exception of project and join which deal with fields, these operations are applicable to sets of any type, including Standard Pascal sets. Relational algebra operators are included because of their benefits demonstrated in conventional databases, in terms of programmer productivity and automatic optimisation. The new operators are selection, desetting, ordering, cartesian product (denoted #), join and projection. Some examples of set expressions are given in figure 2.1. The priority of operators is as follows:

NOT, selection, projection, ordering, desetting : highest precedence;
multiplying operators (* / # join div mod and) : second highest;
adding operators (+ - OR) : third highest;
relational operators (< <= > >= = <>) : lowest precedence.

```
( [1,2,3,4,5] AS S) [ S < 4 ]       is  [1,2,3]
( [1,2,3] - [3,4] * [2,3,4] )       is  [1,2]
[1,2,3] # [4,5]                     is [ [<1,4>], [<1,5>], ..., [<3,5>] ]
[1,2,3] <= [1,2,3,4]                  is TRUE
[1,2,3] < [1,2,3]                   is FALSE
([2,3,4,5] AS E) [| ODD(E) |]       is 3 or 5 (an integer)
FOREACH ([1,2,3,4] ORDER DESC) AS E DO
        WRITE (E:4);                is   4  3  2  1
WRITE( ([1,2,3] ORDER DESC) : (:4)) is   3  2  1
```

Figure 2.1. Examples of set expressions.

2.4.1 SELECT

This operator can be applied to a set of any type, yielding some subset of the original. The notation is:

E1 [E2]

where E1 is a set expression and E2 is a boolean condition. The effect is to evaluate E2 for each element of the original E1; the element appears in the result set if E2 evaluates to true. The type of the result is the same as the type of E1. The identity of elements in E1, and thus the cardinality of E1, is fixed at the beginning of the selection. If E2 calls a function which inserts in or deletes from E1, these changes do not affect the selection.

2.4.1.1 CURSORS

To reference elements within a selection, desetting or projection, the set name is used as a cursor; and even this identifier can be omitted in many cases. This avoids additional syntax to indicate the current element, such as Setname$ (or Setname^ if we wish to overload this symbol).

The scope of a set identifier extends over unary set operators applied to it. A set expression accesses and establishes a reference to each set therein, and then executes these unary operators; the reference exists during the execution of these operators. Within this extended scope, the set variable identifiers are used to reference the current element (in database terms, they act as a cursor). If the current element is a record, its field identifiers can occur in a field designator without respecifying the record name (as in WITH), to denote the appropriate field of the referenced element. For example, if S is a set of records having a field "age", then

S [age < 50]

can be used in place of S [S.age < 50].

If A is a variable and A also names a field of S elements, then the variable A and the set S are inaccessible in unary operators applied to S. Aliases permit references to variable A or to S itself, and are also useful in expressions where the same set appears more than once. An alias is introduced by following a set expression with "AS aliasname". This associates the elements of the expression with "aliasname" instead of the set variable name. Examples:

((X + Y) AS XY) [XY[1] < 9]

(S^ AS E) [E.age < 50]

In the latter example the alias is unnecessary; it is shown as an alternative to conventional cursor usage. The general form of the alias construct is:

AS aliasname (fieldlist)

where either aliasname or the bracketed fieldlist may be omitted but not both. The fieldlist is used for naming the fields of a projection, intersection, union or difference. Such naming is necessary only if the field is to be referenced in a subsequent projection, selection, desetting, ordering or join. Example:

(X * Y) AS (a,b,c) [a < b + c]

which is equivalent to

X [a < b + c] * Y [a < b + c].

EXAMPLES:

Mths [Mths < Nov]; {Mths a set of enumerated values}

Money [Money * 1.1 < Num]; {Money a set of REAL}

Payset [(Payset * SmallAmounts) = []] {Payset a set of sets}

Mths [Mths IN RainyMths [RainyMths > Jun]; {sets of enumerated values}

Mths [(Mths IN RainyMths) AND (Mths > Jun)] {equivalent to the above }

2.4.2 DESETTING

A single element can be extracted from a set using the desetting operator.

E [| predicate |]

extracts an element in the set expression E that satisfies the given predicate. The predicate can be the boolean literal TRUE, if any element will suffice. The type of the result has the base type of E elements and can be used in the same way as any object of this type. When passing a set element as parameter, the desetting is performed when the procedure is called. As in selection, the cardinality of E and the identity of its elements are fixed before the desetting commences, so side-effects in the predicate cannot cause new elements to be considered or remove elements from consideration. Desetting is an operation and so, like all the other set operators described, it cannot appear as an L_value nor be supplied as an actual VAR parameter.

Examples:

```
suppliers^ [| city = 'LA' |]    (* any LA supplier *)
(newparts * baseparts)[| TRUE |] (* any element in both sets *)
suppliers^[|sno=29|].sname      (* name of supplier 29 *)
```

2.4.3 ORDERING

A set can be ordered using the operator ORDER followed by ASC or DESC specifications. If the elements are simple values or strings, no additional information is needed, eg.

```
NUMSET ORDER DESC
```

If the elements are arrays or records, any number of ASC/DESC specifications may follow the ORDER operator, to indicate the primary sort key, secondary sort key, etc.:

```
PARTSET ORDER DESC cost, DESC transport_cost (* field names of partset
    elements *)
ARRSET^ ORDER ASC [1,1].size (* arrset is a 2-dimensional array of records *)
```

Ordering of sets is useful in a FOREACH and WRITE(LN) statement. The compiler issues a warning when ordering is used in a meaningless context, such as L := R ORDER DESC. A field that is part of a variant may be used to specify the ordering desired, but will cause the statement to fail if an element is encountered for which that specific variant is not active. A selection should be used to ensure that only the expected variants are processed.

2.4.4 TIMES (CARTESIAN PRODUCT)

The cartesian product operator, denoted by a # and read as "times", can be applied to sets of any type. The operands can be any (possibly different) set types. The result of an N-way product S_1 # S_2 ... # S_N is another set, having as base type a record with N fields. The type of the i'th field is the base type of the i'th set in the product. The name of the i'th field is the name of the i'th variable; variables are aliased if they appear more than once.

This approach is different from that of relational databases, which deal only with sets of tuples. The result of an N-way product in a relational system is a tuple with (F_1 + F_2 + ... + F_N) fields, where F_i is the number of fields in the i'th operand. This approach is not used in P-Pascal since it does not extend well to other element types. For example in X # Y, where the base type of X elements is Real and that of Y elements is Integer, the elements of the product are records comprising a Real field followed by an Integer field. Similarly, if the base type of X elements is RecTypeA and that of Y elements is RecTypeB, the product elements comprise two fields of type RecTypeA and RecTypeB.

We shall see later that the projection operator allows values to be re-organised into a different structure if necessary.

Examples:

```
(* Construct a set of records where every employee is associated with each value in
        NumSet.  Elements comprise one integer and one employee-record field *)
        (NumSet # EmpSet)

(* all data on employees not located in their own dept:*)
        (EmpSet # DeptSet) [ (EmpSet.dno <> DeptSet.dno)  AND
                                (EmpSet.office = DeptSet.location) ]
(* all data on employees and their managers: *)
        (Emp # Emp AS M) [ Emp.manager = M.eno ].
(* data on any supplier situated at a project site: *)
        (projects # supplierset) [| site = location |]
```

2.4.5 NATURAL JOIN

This operation is defined for sets of records. The result of S1 JOIN S2 is a set of records with fields F1, F2, ..., Fn, F'1, F'2, ..., F'm. Fi are the fields of an S1 element in order, and F'i are those fields of S2 elements (in order) which are not shared with S1. A field is "shared" by the sets if that field name occurs in both base types. Elements appear in the result if they are in the cartesian product of S1 and S2, and additionally have the same values for all shared fields. If shared fields have incompatible types, the join result is the empty set. Since the elements of a JOIN are structured like those of relational systems, the P-Pascal JOIN is identical to the natural join of the relational algebra. If a unary operator is applied to a join, field names can be used directly within this scope, without ambiguity.

Examples:

((X^ + Y) JOIN (Z - W^))

(EmpSet JOIN DeptSet)

(EmpSet JOIN DeptSet) [wage >= 9000]

In the above example, if wage is a field of EmpSet elements only, this expression is equivalent to

(EmpSet [wage >= 9000] JOIN DeptSet)

If it is a field of both sets' elements, it is equivalent to

(EmpSet [wage >= 9000] JOIN DeptSet [wage >= 9000])

If S1 is a set of [<a, b, c>] records, and S2 is a set of [<e, c, d, a>] records, then

(S1 # S2) [(S1.a = S2.a) AND (S1.c = S2.c)]

is a set of [< [<a,b,c>], [<e,c,d,a>] >] records, while

S1 JOIN S2

is a set of [< a, b, c, e, d >] records.

2.4.6 PROJECT

This operator permits the construction of records by giving the value of each field in turn, using a sequence of expressions. Their order defines the field ordering of the result. This is significant for type compatibility, and when the projection occurs as a READ(LN) or WRITE(LN) parameter. Array, record and set aggregates can be used to construct records of any type. Projection can be applied to single records, or to sets or arrays of records. A projection can reference its operand as well as other data.

The syntax is E -> (E_1, ..., E_n) where E is the record(s) being projected over and E_1, ..., E_n is the field list. As with selection and desetting, the cardinality of E and the identity of its elements are fixed before the projection is applied, and cannot be affected by changes in the E_i's. Named and unnamed projections are distinguished. A named projection is one where each E_i contains a single identifier, and no two E_i's contain the same identifier. The field names of the result record are given by these identifiers. If a projection does not have this property it is an unnamed projection; where it is necessary to name any field(s) thereof, an alias is used. Example:

X ->(a, b, c+10) JOIN (Y ->(a, a, b+c, 24) AS (a,d,e,f)

The fieldnames in the first join operand are *a,b,c* and in the second operand are *a,d,e,f*.

Project is useful for two main purposes: creating arbitrary record structures and updating set elements. This is illustrated by means of examples.

In

SP := PS -> (pno, sno, disct)

if PS is a record having fields *pno, sno, disct, cost* and SP is a record having fields *pno, sno, disct*, the *cost* field is eliminated by the projection. If PS and SP were sets or arrays of these records, this would be done for every element. A projection onto a set causes duplicates to be eliminated. For example, if *sno,pno* is a PS key, the projection

PS -> (sno, disct)

will contain one record for each *sno* value - the first encountered with that value. Thus this type of operation should only be performed if the program is looking for any *sno-disct* combination. If it requires a specific *sno-disct* pair, the corresponding selection should be given.

A projection can be used to "reformat" the result of a product, eg.

(SP # PJ) [pno = pnum] -> (SP.sno, SP.pno, PJ.jnum)

Records having any kinds of fields can be constructed using aggregation, eg.

X -> (a, [< b,c >], [. 0,0,0,0,0 .], [e,f])

constructs a record with a record, array and set as its last three fields.

Projection provides a simple and natural way of updating record collections, as shown below.

SP := SP -> (pno, sno, disct + .05)

If SP is a set or array, all its elements are altered so that "disct" is increased by 0.05. If SP is a record, this statement is equivalent to SP.disct := SP.disct + .05. Conceptually, a projection is applied to all elements simultaneously. For example, if sno is a key of S then

S := S -> (sno + 100, city, telno)

does not alter the cardinality of the set.

2.5 STATEMENT CHANGES

2.5.1 THE FOREACH STATEMENT

This is the only new statement in P-Pascal. It iterates over the elements of a set or array expression, termed the "control". The name of the control acts as a cursor to reference the current element within the loop; and if this is a record type, this name can be omitted in field designators. The identity of the elements to be iterated over is fixed upon entry to the loop. Thus where the control is a set, its cardinality is fixed, so that insertions and deletion in the repeated statement do not affect the control; in particular, infinite iterations cannot arise. Array iteration proceeds in row major order. The order of iteration over sets [Atki87a] is either random but repeatable (if there is no ORDER on the control), or as defined when the iteration begins (if the control is ORDERed).

There are no restrictions on the statements which can appear within the loop. When the control is a variable the loop body may assign values to the iterator elements. If it is a set variable to which unary operators are applied, the right to assign values to its elements is retained. If the loop changes elements in the control, the changes will be reflected in elements after the loop. The statement is thus useful for updating subsets, as in:

```
FOREACH S [ status = x ] DO S.value := S.value + 10;
```

If the loop statement deletes or inserts elements into a control set, this will not affect the iteration. That is, the identification of which elements are in the control set is fixed on loop entry; only their contents may subsequently be changed. Duplicates are removed from a control set on loop termination. For example, if S had keys 101,102,201,202 initially, then

```
FOREACH S DO S.key := S.key + 100;
```

leaves S with keys 201, 202, 301, 302 after the loop; while

```
FOREACH S DO S.key := S.key DIV 10;
```

leaves S with keys 10, 20 after the loop.

2.5.2 INPUT/OUTPUT

The READ(LN) and WRITE(LN) procedures of Standard Pascal are extended to admit record, array and set parameters. Each field value of a record is read from the input file, and assigned to fields in the order in which these occur in the record declaration. Pointer, file and enumerated fields are ignored, since their values cannot be input. Writing of

records is handled analogously. In reading an arrays of N elements, the next N values are taken from the input file and placed in each element in turn (row major order); and similarly for writing. To input the value of a set, a sequence of values enclosed by symbols [and] is required in the input file. If "]" is required to be in the set, a double bracket "]]" is used instead. If duplicate keys or duplicate elements occur, only the first such element becomes part of the set. When a set is written, its elements are output without enclosing brackets.

Bracketed format specifications can be given for structured data. This is applied repeatedly until the printable values in the structure (i.e. excluding enumerated types, pointers and files) have all been output. If the formatting for a variant record causes REAL formatting to be applied to a non-REAL value, a runtime error results.

EXAMPLES:
WRITELN (intarrptr^ : (:5, :10), rec_o_nums : (:5:2, :5), intsetptr^ : (:10));
READ (somerec, a_char, set_of_arrays);

2.6 PERSISTENCE

The P-Pascal persistence mechanism is based on that of PS-Algol[Atki83], which has the following properties:

- The persistent store is shared by many databases, able to cross-reference each other.
- Only heap objects can persist, and database data is accessed via pointers.
- Functions are introduced to save and locate "primary" objects (named objects explicitly placed on a database); all objects referenced from these are automatically preserved.
- Commit causes all database changes effected by the program to be applied to the persistent store in an atomic action, and it is possible to recover from Commit failure.
- Programs need only give type declarations for those structures which they will be processing, and not for the entire database.

Some differences from the PS-Algol approach to persistence specification exist. In particular, there is no table construct. Primary objects are not restricted to being records, but can be of any type, eg. simple values, sets or arrays, and the representation of the

primary-name to primary-object mapping is transparent. Databases can optionally be clustered. Read and/or Write passwords can be specified with type declarations, database operations, and the "primary" operations Save, Drop and Locate. Metadata can be used within a program to discover type names, definitions, security, owners, cross-references and extents. These aspects are dealt with in chapters 5 through 7.

2.6.1 PERSISTENCE FUNCTIONS

These functions are:

1. CreateDB (db, databasename, owner, placement, setfactor, metadata_flag, readkey, writekey)
2. OpenDB (db, databasename, mode, readkey, writekey)
3. Save (db, primaryname, pntr, readkey, writekey, part_keys)
4. Locate (db, primaryname, pntr, readkey, writekey, part_keys)
5. Drop (db, primaryname, readkey, writekey, part_keys)
6. Commit

where

databasename, readkey, writekey, primaryname and owner are strings;
db is pointer variable of standard type DBPTR;
placement and setfactor are integers indicating the clustering required for a new database;
mode is "R" for read or "W" for write;
pntr is a pointer;
metadata_flag is a Boolean
and part_keys is a set of strings.

Save stores the referenced primary object on a database and associates it with a string name. Locate fetches a primary object by giving its name and a pointer variable into which the address of the retrieved object must be placed. Drop removes the primary status of an object; it will not be deleted from the store while any other persistent object references it. Commit updates all open databases to reflect changes made by the program since the last Commit, in one atomic action. The remaining persistence statements are self-explanatory.

The *pntr* argument of persistence functions can be any pointer type, since objects of any kind can be read from and written to a database, just as different types can be read from

and written to files. The *db* parameter points to a unique object in the database known as the root of persistence (described in the next chapter); however this is hidden from programmers, to whom internal database organisation is transparent. Thus DBPTR is a standard type, but the type of the target object to which it points is hidden; programs are restricted to using the pointers and cannot access the data they reference except through Save, Locate and Drop. The role of the metadata_flag, readkeys and writekeys will be elaborated in subsequent chapters. All six functions return a pointer to a standard error record, or NIL on successful execution.

2.7 CONCLUSION

Requirements for Bulk Types at the language level are given in [Atki89] as:

1 The size of the type definition is independent of the size of the value eg S: SET OF RECORD a:b; c:d END.
2 There should be a well defined, convenient algebra over the values of these types.
3 Associative access operations should be provided.
4 Iteration over the elements of a value of such a type should be supported.
5 It should be possible to specify the order of such iterations.
6 The constructors should be data type complete.
7 Understandable definition and efficient implementation of type checking is needed.
8 These types should be consistent in design and use with others in the language.
9 Efficient notation for values of such types is needed.

We briefly consider how P-Pascal meets each of these requirements in turn. Set definitions are succinct, particularly as the elements must be of named type. The relational algebra is well defined, and known to be a convenient algebra over sets [Codd]. Associative access exists in the form of the selection operator, and set types implicitly provide an index type as a special case. This arises when elements comprise two parts: a key, and a pointer to an object. These types can also be used for composing objects into multiple overlapping sets without causing data replication, as only the pointers are duplicated. Iteration is provided by the FOREACH statement, and can be sequenced by applying the order operator to the control set. P-Pascal sets are data type complete, and type checking aspects are considered in chapter 6. The language has been designed to conform to Standard Pascal wherever possible, and the set type has been treated consistently with other types. For example, the iterator applies to sets and arrays,

projection applies to records in any context and not only sets of records, cartesian product is defined uniformly for sets of any type, etc. In addition, the Standard Pascal notation for set values has been retained, and extended to the other type constructors of the language; fortunately this syntax is indeed efficient.

The provision of a type-complete set constructor overcomes the limitations of record-based systems identified by [Kent]. These can be broadly categorised into three kinds. Firstly, the correspondence between first normal form records and entities is not one-to-one. In P-Pascal all the attributes of an entity can be stored in a single record, including multi-valued properties. Moreover, references to other objects are not constrained to key usage; objects can be directly nested within each other, or pointers can be used. This avoids problems caused by multiple alternative keys and missing key values. Finally, where entities of the same type can have different properties according to their role, this can generally be accommodated by means of variant records. An exception occurs when an entity can have the attributes of two or more variants: generalisation (or inheritance) goes further towards resolving this.

The language introduces persistence and type completeness to Pascal, while minimising the changes and extensions to the language. This approach can be contrasted with other suggestions for improving Pascal, where foreign constructs such as private variables and sequencers [Tenn77], or exits and classes [Tenn83] are proposed. Other suggestions for improving the language, some of which were included in the ISO Standard, can be found in eg. [Habb], [Knob], [Leca], [Wels77], [Tenn78] and [Henn], while [Tenn77] illustrates how the principles of correspondence and abstraction can be incorporated.

The following goals for persistent languages were identified by the Napier88 developers [Morr89b]:

- controlling the complexity of the system so that developers and users can concentrate on the applications. This is achieved through the use of consistent rules throughout the system and the introduction of a minimum of new concepts.
- protection of data from misuse and failure.
- orthogonal persistence: the use of data is independent of its persistence.
- a mechanism for controlled system evolution.
- concurrent computation.

P-Pascal provides orthogonal persistence and introduces a minimum of new constructs into Pascal, namely the high-level set operators permitting non-procedural manipulation. These operators make data collections easier to access and enable automatic optimisation of expressions. Protection is provided in the form of strong and eager type-checking, and programmer-defined security constraints. As advocated in [Atki87a], a program can be used against different databases and these can be determined dynamically according to its input. Attention has not been given to data evolution or concurrency control. P-Pascal databases are unique in that sets and graphs can be intermingled within a single data structure. Since it integrates two well known disciplines - Pascal programming and relational database manipulation - it should be easy for many to make the transition to persistent languages through P-Pascal.

CHAPTER 3 DATABASE PROGRAMMING LANGUAGES

Persistent languages are a special kind of database programming language [Atki85,Dear89]. This chapter describes the most well known examples and compares their approach with that of P-Pascal. In conclusion, some language design criteria are presented.

Database programming languages can be broadly classified into three classes, according to their underlying data model. Relational languages are those having the equivalent of a relation as bulk data type, with relational algebra or relational calculus operators. Conceptual languages have a semantic data model as their basis. Persistent languages extend a general-purpose programming language by incorporating persistence as an orthogonal property of all data types. A distinction given in [Dear89] is that persistent languages are program-centred rather than data-centred, holding the view that programmers write procedures to operate on data, rather than treating functions as data properties.

Relational languages include Adarel [Horo], Aldat [Merr], Astral [Ambl], DBPL [Schm88b], Modula/R [Koch], Pascal/R [Schm77], PLAIN [Wass], RAPP [Hugh], RIGEL [Rowe] and Theseus [Shop]. The most well known conceptual modelling languages are Adaplex [Smit83], Galileo [Alba85a], Machiavelli [Ohor], Quest [Card88] and Taxis [Mylo]. Languages in the third class include Amber [Card85a], E [Rich89b], FAD [Banc87], Gemstone/Opal [Maie], Hope+ [Perr], Leibniz [Ever], Modula-3 [Card88], Napier88 [Morr89a], O++ [Agra], Oberon [Wirt88], Pascal/M [Rose89], Persistent Prolog [Gray], PGraphite [Tarr90], Poly [Matt], PS-Algol, Staple [Davi] and *X* [Saje]. This chapter describes one or two languages of each class. Those selected are the seminal works and the most well known examples. Where appropriate, languages closely-related to P-Pascal have been chosen.

3.1 PERSISTENT LANGUAGES

This approach extends a conventional language by incorporating persistence, without introducing any database model. The first such language, PS-Algol, is described here, as well as its successor, Napier88. These languages are chosen because they are the most well known, and the P-Pascal implementation is based on their approach.

3.1.1 PS-ALGOL

3.1.1.1 INTRODUCTION

PS-Algol [Atki81, Carr87] was the first persistent programming language to be developed. It introduced orthogonal persistence into the language S-Algol [Cole]. S-Algol was chosen as a vehicle for investigating persistence because of its power achieved through simplicity and consistency. In addition, its use of pointers to represent structured data permits incremental binding; and its untyped pointers permit dynamic type-checking. The principle of orthogonality guided the development of PS-Algol : all data types have equal status in the language. Thus data of any kind can persist, and type constructors obey the principle of data type completeness, enabling a rich type system to be described with a small number of rules.

3.1.1.2 DATA TYPES

The data types of PS-Algol are integer, real, boolean, string, file, vector (array) and structure (record). In addition there are data types for graphics, which will not be considered here. The language is type complete, with the result that structure and vector components can be of any type, including the type proc (procedure or function). Any object can optionally be declared to be constant, in which case it retains its initial value assigned at creation time. Uninitialised objects are not permitted.

Vectors and structures are heap objects and their copy semantics are pointer based. The type "pntr" is an untyped pointer that references a structure. The same pntr variable or component can point to instances of different structure classes at different times. The type of a vector having elements of type T is *T (a pointer to a sequence of T's). All objects of type *T are type compatible, irrespective of their upper and lower bounds.

The original version of PS-Algol was extended to include the "table" construct, not present in S-Algol. This was found to be desirable for managing large data collections. A table is logically a collection of <string,pntr> pairs and <integer,pntr> pairs, representing a collection of "named" structures. The only means of manipulating tables is through three standard procedures, for insertion/deletion, iteration and associative retrieval.

Untyped pointers provide a degree of polymorphism and an opportunity for type evolution. If structures contain a dummy PNTR field, this can later be replaced by a reference to new information. However too many levels of indirection, caused by frequent data structures additions, can become cumbersome.

3.1.1.3 OPERATORS

The operations defined on tables have two different forms each, one for string keys, the other for integer keys. They are S.Enter (I.Enter), S.Lookup (I.Lookup) and S.Scan (I.Scan). Enter accepts a key and a pointer to the structure having this key. The lookup functions return a pointer to the structure with the given key (else NIL). The Enter operations are also used to delete entries by associating a key with the NIL pointer. In a scan, a user procedure is applied to each table entry in turn until it returns false, or the table is exhausted. The scan returns an integer indicating the number of times the procedure was invoked.

3.1.1.4 PERSISTENCE

In PS-Algol there is a single persistent store which is shared by any number of databases. These are able to cross-reference each other; in general it is impossible to know on which database an object will be stored. A database comprises a collection of named objects, which are explicitly inserted into a database, and the transitive closure of their references is retained in a database. That is, reachability is employed as persistence mechanism. Only heap objects can persist, and database structures are only accessible via pointers. The Commit statement causes all database changes effected by the program to be applied to the persistent store in an atomic action. Until then, the databases are left intact.

Practical experience clearly showed that tables are heavily utilised [Carr]. In particular, the root of persistence in a database is a table. A consequence of this is that all database objects are of structure or vector type. Thus in order to save a single object (eg. a Procedure) on a database, it must first be "wrapped up" in a structure or vector.

3.1.1.5 COMPARISON WITH P-PASCAL

In PS-Algol, data is not as precisely described as in P-Pascal, since pointers are untyped. In addition, problems with the naming system result. The field names in any scope must be unique, as the type of a referenced structure is determined on this basis. Field names therefore tend to be unique across the entire persistent store, which is tedious for programmers and makes merging difficult [Atki87b]. Since P-Pascal pointers are typed, there is far less dynamic type-checking than in PS-Algol and field names need not be unique across the persistent store.

The language presents an indexing mechanism directly, whereas P-Pascal provides this through its set type. The bulk data type and operators introduced into Pascal differ from the table construct, which is designed for procedural access. Key specification is less restrictive and elements do not have to be records. PS-Algol has fairly low-level data types, which provide flexibility but can require much detail in programs [Atki84].

3.1.2 NAPIER88

3.1.2.1 INTRODUCTION

This language is the successor of PS-Algol. As such it retains PS-Algol features which were found to be successful, and extends the language in several ways, reflecting the current trends in programming language design. Type-checking is partly static and partly dynamic; the programmer chooses the appropriate type constructor accordingly. There are existentially quantified types (abstract data types) and universally quantified types (polymorphic procedures).

3.1.2.2 DATA TYPES

The scalar types are those of PS-Algol: integer, real, boolean, string, pixel, picture, file (and null). Types image, vector, structure and proc are also retained. Data type completeness, structural equivalence and incremental binding are employed. The PS-Algol types are extended to include polymorphic procedures, variants, environments and abstract data types.

Napier88 refines the dynamic binding and type-checking of PS-Algol, in that it does not limit this to one type PNTR, and makes the binding time explicit. Types ENV(ironment) and ANY are dynamically checked. Environments constitute name : value bindings which can be dynamically composed. Type ANY is the type of the union of all values in Napier88. Values must be injected into and projected out of "any". A variant is an object which can have one of a list of possible values at any point in time. The values in the list can be of different types, and there is no associated tag determining which variant is active.

An abstype (abstract data type) has an interface which is a list of elements defined in terms of abstract witness types. Elements can be any kind of Napier88 object, including proc(edure). If two objects are defined in terms of the same abstype but with different concrete witness types, they are of the same type. Abstype elements are referenced in the same way as structure components. An abstract data object is only equal to itself, i.e. equality is identity, and it may only be used with its own operations. Parameterised types provide a shorthand for declaring similar types, and are most useful with recursive type definitions.

3.1.2.3 OPERATIONS

As in PS-Algol, uninitialised objects are not permitted and copy semantics is pointer based. New operators are incorporated to support the new data types; for example IS and ISNT indicate if a specified variant is active or not.

The PROJECT operator enables a value injected into ANY to be processed according to its type. It prevents side effects by binding the value to an identfier, and allows for static type checking in the statements that follow. The USE clause permits static checking of abstract data objects through a constant named location, in a similar way. The witness type can follow this identifier, enabling it to be employed in the scope of the USE clause.

USE <clause> AS <identifier> [<witnesses>] IN
    <clause>

This is the only way an abstract object can be dereferenced; it is rather restrictive, but this is unavoidable if static type checking is required.

3.1.2.4 PERSISTENCE

Environments are introduced as a means of modularising and controlling the name space. The standard procedure "environment" creates a new environment into which any number of bindings may be placed or removed. Environments are also used to dynamically compose block structure, as specific bindings in an environment can be brought into scope with a USE clause. The existence of a binding in an environment can also be tested. Napier88 goes a long way towards supporting type evolution, with the exception that the interface of abstract types cannot change [Atki87b].

The root of persistence is an environment named PS. All data reachable from this environment is made persistent. Embedded environments induce a structuring of the store. A new environment created in PS will only persist if the standard procedure Stabilise is invoked, or the program terminates normally. During a stabilise, the current state of all open files is also saved in the persistent store. When the Napier88 system restarts, the state of these files is recovered.

3.1.2.5 COMPARISON WITH P-PASCAL

Napier88 lays emphasis on programming language aspects in its type system, investigating concepts such as polymorphic procedures and abstract data types in a persistent environment. In contrast, P-Pascal aims at introducing a minimum of new constructs into Pascal, to study the incorporation of database facilities in a persistent language - viz. bulk data manipulation, metadata maintenance, security enforcement and physical clustering.

Representing the root of persistence as an environment, together with the ability to dynamically add bindings like nested environments, provides a simple and powerful means of dealing with a structured persistent store. This approach is neater than the use of Save/Locate functions and inter-linked databases.

2.1.3 OTHER PERSISTENT LANGUAGES RELATED TO P-PASCAL

Pascal/M [Rose89] introduces orthogonal persistence into Pascal. The central themes of this language are information-hiding [Parn] and the use of a capability-based machine architecture [Rose85]. Pascal is thus extended in two directions: modules (organised into

classes) and capabilities (access rights), which are both first class entities in the language. The data types of Pascal are otherwise unchanged; in particular, sets are not type complete.

3.2 RELATIONAL LANGUAGES

These are languages which incorporate the relational database model into a conventional, general-purpose programming language. Examples described below are Pascal/R [Schm77] and its successor DBPL[Schm88b]. These are chosen as they are arguably the best and most well known examples of this class of language; Pascal/R was also the first Database Programming Language developed. Others closely related to P-Pascal are briefly described at the end of this section.

3.2.1 PASCAL/R

3.2.1.1 INTRODUCTION

This was the first project to incorporate permanent data objects within a conventional programming language. It extended Pascal with the data types relation and database. Pascal/R was developed to extend an algorithmic language with constructs for efficient processing of large amounts of inter-related data. In particular, high level statements were considered essential in order to permit automatic optimization of database accesses [Schm77]. It was also intended to act as a vehicle for further development of the relational model through, for example, the incorporation of programming language concepts like abstract data types. The main problems in designing this language were how to integrate the two different types systems, how to identify databases and introduce database names into a program and deciding the data manipulations to support [Atki87a].

3.2.1.2 DATA TYPES

Tuples are represented by the Pascal Record type. The type Relation defines a set of tuples of the same record type. The fields of these records are constrained to be scalars or strings, so that the component values are atomic and the relations are in First Normal Form [Codd]. A single key, or unique identifier, is defined for every relation. Temporary relations are treated identically to persistent ones.

3.2.1.3 OPERATORS

A relation iterator is introduced:

FOREACH control_rec IN range_rel DO statement.

Statements manipulating relations are different from other Pascal statements, being closer to the traditional database operations of insertion, deletion, update and initialisation. These are respectively denoted by:

R1 :+ R2

R1 :- R2

R1 :& R2

R1 := [rec]

In R1 :+ R2, those tuples of R2 whose key value are not in R1 are added to R1. The other statements operate analogously. The R-value of a set assignment can only be a single relation or relation constructor. Record aggregates are introduced, permitting (for example) tuple insertions like

R1 :+ [< 312, 'AJAX', 'LA', 10 >] ;

Relational constructors produce some restriction (selection), projection, product and/or join of relations. They are defined using the relational calculus [Codd], and predicates can include existential and universal quantifiers. Schmidt identified problems with constructed relations, such as the definitions of their keys and the possibility of checking keys at compile time [Schm77].

3.2.1.4 PERSISTENCE

A Database construct introduces names from the database into the program. It is conceptually equivalent to a record having all its fields of type Relation. Databases are listed in a program heading in the same way as files. This potentially allows the program to use different databases of the same type on different executions. Variables of type Database are persistent. To access part of a database, such a variable is used within a WITH statement.

3.2.1.5 COMPARISON WITH P-PASCAL

P-Pascal differs considerably from Pascal/R, which integrates two different systems into one language. In particular, P-Pascal is type complete and the full range of persistence is

possible for all data types. No additional types such as relations are introduced, and sets are an extension of Pascal/R's normalised relations that permit elements of any assignable type. Multiple keys can be defined, giving greater flexibility in terms of set access and expression optimisation.

In P-Pascal the relational primitives are introduced in the form of new set operators, rather than the Pascal/R set constructor. This was done in order to make the resulting syntax more concise and Pascal-like. Pascal/R bulk operations are based on the iterative control structures of Pascal. P-Pascal uses the existing Pascal set type and thus introduces bulk operations using a notation similar to the other (infix) set operators. This seems more natural than modelling the new operators on control statements. Statements operating on sets are not distinguished from other Pascal statements. The conventional assignment statement can be used to assign any set expression to a set variable. This is unlike Pascal/R, where statements :+, :- and :& are introduced to represent insertion, deletion and updating; and the R-value of a set assignment can only be a single set (or set constructor). Pascal/R is based on the relational calculus rather than relational algebra. Intersection, union and difference cannot be applied to its relations, and natural joins must be specified in full as a restricted product. Universal and existential quantifiers are supported; P-Pascal instead permits comparison of sets by means of containment operators. Pascal/R also lacks desetting and ordering operations.

An entire Pascal/R program execution is seen as a single transaction, and can operate on only one database. The language also does not permit dynamic selection of this database. As databases cannot be declared in a procedure, programmers are forced to learn about a database; they cannot simply use a database procedure library [Atki84].

3.2.2 DBPL

3.2.2.1 INTRODUCTION

DBPL[Schm88b] is the successor of Pascal/R. It extends Modula-2 with the types relation, selector and constructor, as well as transactions and database modules, allowing almost any type of value to persist (but not pointers).

3.2.2.2 DATA TYPES

The data type RELATION is a structure comprising elements of the same type. Any type may serve as the base type, except pointers, which are not first class objects and cannot be made persistent. Relations of records or arrays can have a single declared key, which is a unique identifier for its elements [Schm88b]. The key can comprise any number of values, of simple or string type, and cannot include part of a variant record section. Variables of procedure type can be declared, and structures can include procedure components. Recursive types are not supported.

Selectors and constructors are first class objects, denoting an "access expression" or rule for accessing relations. The data type SELECTOR stores a predicate on some single relation, and defines a subset thereof. This is analogous to the restriction operator of relational algebra. Additionally, access restrictions can be associated with selectors, to limit the operations that may be performed on the selected data.

Access restrictions in selectors define the operations applicable to selected relation variables. If a selected relation variable is passed as a variable parameter, it must have no access restrictions. The operations that can be restricted or permitted are those updating, inserting, deleting and assigning to the selected relation variable. A fifth access right permits the variable to be used in expressions, FOR EACH statements and standard procedures. An unparameterized selector names a selective access expression on some global relation variable - i.e. it names and protects some subset of that relation. An ON clause allows any relation variable of the given type to be specified when the selector is used. The WITH section enables parameters to be used in the condition defining the subset.

A CONSTRUCTOR is similar to a selector, but defines a relation derived from any fields of any number of relations. It permits projections and joins to be described. Recursive constructors are supported. Variables of selector (constructor) type assume a selector (constructor) as value. An unparameterized constructor names a projection over some product of global relations. Depending on its body, it may represent only a projection of a single relation, or a join of any number of relations, etc. WITH and ON parameters are used analogously to their selector counterparts.

3.2.2.3 OPERATIONS

In addition to the conventional assignment statement, three statements for relation update are provided. Denoted :+, :- and :& they are used in to insert, delete and update elements respectively. Two relations are defined as equal based on equality of key components. All six relational operators may be applied to relations to test equality and inclusion. The set membership operator IN is also applicable. Equality of (variant) records or arrays cannot be tested [Schm88b] and the set operators union, intersection, difference and symmetric set difference cannot be applied to relations.

Record and array aggregates are supported and must be preceded by the type name. The values must be given in order; in the case of records, the order in which the fields were declared in the record definition is required. A similar notation is used for relations; and access expressions, selectors or constructors can define the relation contents. Some examples [Schm88b, Matt89]:

```
parts := Items{EACH p IN oldparts : p.price > k};
parts :& Items{thispart};
oldparts := Items{};
Rel := Reltype{ { f1, ..., fN} , { f, ..., fm} }
```

Access expressions are first order predicates, and can also be used as the control of a FOR loop if unparameterized, for relation iteration. The same syntax is used for identifying a single array or relation element - either index(es) or key component value(s) are enclosed in square brackets. Furthermore $R[E_1,...,E_n]$ is not the only means of desetting; to extract a single relation element, an appropriate selector can also be used.

Opaque types are provided by Modula-2's definition modules and implementation modules. Selectors can also be used to impose constraints, as shown below [Schm90].

```
SELECTOR ConsistentParts : PartRel
BEGIN <assertion> END ConsistentParts;
```

This makes the assignment

```
[ConsistentParts] := OldParts;
```

equivalent to

```
IF <assertion>
        THEN Parts := OldParts
        ELSE exception END
```

3.2.2.4 PERSISTENCE

Persistence is an attribute of a module, as persistent variables are introduced by declaring them within a database definition module. All binding and checking is performed at compile time. A database is defined as the collection of persistent variables in a database module. The initialization section of such a module is executed once only in the lifetime of that database. Database relations can be hidden, forcing clients of the module to use exported selectors (views).

Persistent variables can only be accessed from within transactions. These are procedures defined as transactions in their heading. Transactions are the basis of the concurrency mechanism: no other program can access a transaction's variables while it is executing. They also provide atomicity, and an intention list can be used to indicate which persistent variables are accessed by the transaction. Nested and recursive calls to TRANSACTIONS are not permitted, and the language is statically typed and thus does not lend itself to data evolution.

3.2.2.5 COMPARISON WITH P-PASCAL

There are understandably many similarities between P-Pascal and DBPL, as they have similar goals in extending similar languages. However, since DBPL retains many of the features of Pascal/R, the same distinctions between P-Pascal and the ideas of Schmidt can again be made. In P-Pascal, unlike DBPL:

- No additional type constructor like "relation" is introduced [Schm88b].
- The language is based on the relational algebra rather than relational calculus, and thus bulk data operators include intersection, union, difference and join.
- Set ordering is supported.
- Relation-specific update statements have been avoided; the R-value of a set assignment is not restricted to a single set or set constructor/selector.
- Existential and universal quantifiers are omitted.

Key equality is used in place of element equality in all DBPL operations on relations. This means, for example, that $R < S$ (relation R is a subset of relation S) can be true even if the elements of R are not found in S. P-Pascal instead compares entire elements in intersections, unions and differences as well as in relational expressions. As the notation is derived from the relational calculus, specification of selection, projection and

join is more verbose than in P-Pascal. On the other hand, the syntax for relation construction and desetting fits neatly into the language, being based on the notation for aggregates and array element identification respectively.

Constructors are relationally complete expressions [Codd] and provide, through recursion, the power of deductive databases [Matt89]. P-Pascal is also relationally complete and permits recursion by means of set-valued functions. It does not have selector or constructor types; that is, it does not directly support views. The language could clearly be extended to include such a facility; however, expressions would not be constrained to set variables and view variables, as in DBPL. Arbitrary set expressions can be formulated using any combination of operators; the result type does not have to be declared in order to do so.

Access rights may be associated with DBPL selectors, i.e. with single-relation views, and take the form of read-access, insert, delete, update and assignment privileges. View restrictions protect the data from all programs that might want to access it. The P-Pascal security mechanism differs in that it is applicable to all data types, it distinguishes only two modes (read and write), and authorisation can differ from program to program depending on security clearance. If a selected variable is passed as VAR parameter in DBPL, it must have no access restrictions [Schm88b]. P-Pascal permits an object to be passed as a variable parameter as long as any component thereof is writable.

DBPL has successfully exploited the modern programming language constructs introduced by Wirth when developing Modula-2 from Pascal. Modules provide opaque types and can be used to hide database variables, forcing clients of the module to use exported selectors (views). However, modules are not first class objects, and the language is not truly type complete: functions can only return a simple or pointer type, the equality operator is not applicable to records, variants or arrays and updating of relations elements is restricted to alteration of non-key values [Schm88b]. Pointers cannot be made persistent, so a database cannot contain tree or graph structures, and associations between objects can only be represented by means of keys. The DBPL relations are therefore more restrictive than P-Pascal sets. Database modules are a convenient mechanism for structuring and protecting a persistent store, and transactions provide for concurrency and atomicity.

3.2.3 OTHER RELATIONAL LANGUAGES RELATED TO P-PASCAL

The two examples above are both based on the relational calculus. This section looks briefly at languages using relational-algebra [Codd] instead. PLAIN [Wass] aims at introducing relations into Pascal in such a way as to keep the extensions as simple as possible and to minimise differences from the standard constructs. The language was designed to permit a knowledgeable programmer to control the way data is manipulated, by restricting the complexity of expressions, and placing the onus of secondary index maintenance on programs. Intermediate relations or "markings" are directly accessible. Relational algebra operators are provided, but the combinations in which they may be used are very limited. A FOREACH iterator replaces the FOR loop and can be applied to relations, sets, arrays or markings. Special operators exist for relation insertion and deletion. There are no variant records, pointers are not first class objects and relation elements must be records. In addition to relations, the type system of Pascal is extended in several ways, eg. dynamic arrays, parameterised types, exceptions and modules are introduced.

Theseus [Shop] is a similar language, but aims to serve as a vehicle for studying global optimisation of database programs. Relations are limited to first normal form [Codd] and keys are used to reference objects, rather than pointers. Iteration is intended to be the main retrieval technique and expressions are limited with the aim of allowing only reasonably efficient statements. Relation assignment has copy semantics on the assumption that an optimising compiler will avoid unnecessary copying. Selection, desetting, comparisons and a limited form of projection (constructing only a set of 1-tuples) are the only relation operators besides insertion and deletion.

RAPP [Hugh] introduces relations and relational algebra operators, abstract domains and concurrency into the multiprocessing language Pascal Plus [Wels]. It aims at providing skilled programmers with tools for improving efficiency rather than implementing a large, complex query optimizer. As such it provides low-level relation operators based on the existing file access mechanisms of Pascal Plus: buffer variables are utilised, and in particular are the only way of inserting and updating elements. Indexes are accessible to programs and are very similar to direct access files, except that they are automatically maintained for the associated relation. Relation operators are limited to selection, projection, natural join, product, intersection, difference, union and membership test; and the complexity of an expression is severely limited. Although persistence independence

is provided, the language is not type complete as relations are restricted to first normal form.

P-Pascal differs from these languages in that it is less restrictive and keeps lower-level constructs such as indexes and intermediate results transparent to programs. It introduces far fewer changes into Standard Pascal, and does not limit sets to first normal form. Differences in set operators are also evident, such as the omission of database-like insert and delete operations.

3.3 CONCEPTUAL MODELLING LANGUAGES

These are database programming languages based on some semantic data model. The example described below is Galileo [Alba85a, Alba85b, Alba89a, Alba89b], which has attracted the most interest.

3.3.1 GALILEO

3.3.1.1 INTRODUCTION

This language introduces semantic modelling features and programming language abstractions into ML [Miln]. It is an expression-based, interactive language. The system accepts an expression, executes it in such a way that the effect is preserved on the database, and displays the result to the user. It differs from other conceptual languages like Adaplex in that the new features are not limited to databases. Galileo provides a modularisation mechanism permitting incremental design and testing, and application-oriented database views. A formal description of the language is given in [Alba88].

3.3.1.2 DATA TYPES

The type constructors of the language are tuples, functions, discriminated unions (or variants), modifiable values, abstract types and sequences. Galileo is strongly typed, and all type checking is static. Recursive types are fully supported. Sequences, rather than sets, are the bulk data type. Functions are first class objects and can be part of structured objects, be returned as values, etc.

Abstract types are introduced for protection through encapsulation and also for tailoring operations for each type. Structural equivalence is used for concrete types, but name equivalence for abstract types. A semi-abstract type is different from any other defined type, but is manipulated using the operators of its representation type. Procedures to create new occurrences by mapping the representation type into the semi-abstract type are provided automatically. A (completely) abstract type hides the operators defined for its representation type, restricting access to the application of functions in the abstract type declaration. Abstract and semi-abstract types can optionally include assertions restricting the values instances may hold. These can be named so that failures can identify the exact assertion that was violated.

3.3.1.3 SEMANTIC DATA MODEL PRIMITIVES

Conceptual modelling abstractions form the nucleus of Galileo: classification exists in the form of classes (abstract data type extents); aggregation is fundamental in the data model (since associations are represented directly, rather than by references) and generalisation is supported (subtype hierarchies and inheritance).

Classes are sequences of (semi-)abstract data types. They are used to represent bulk collections of inter-related objects. Unlike other sequences, classes can optionally have any number of keys defined; if none is given, duplicate but distinct elements can occur. All objects of the abstract type automatically become members of the class when they are created. The remove function excludes the target object from all classes to which it belongs, independent of its appearance as components of other class elements. Elements of classes are aggregates since they can include objects of other classes as components, to represent relationships. This is implemented using data sharing, rather than by means of keys. Elements are uniquely represented, so if an element is updated, this is reflected in all objects in which it is a component.

Derived, optional and unmodifiable attributes are modelled, as well as default values. Subtypes are automatically inferred for concrete types, and can be explicitly stated for abstract types. The type hierarchy distinguishes three kinds of subclass:

subset classes : when a subset of the parent class has been explicitly included

partition classes : which together partition the parent class (i.e. no member of the parent occurs in more than one of the partition subclasses)

restriction classes : which are defined by some predicate (condition) on members of the parent class.

A subclass definition can differ from that of its parent class in that attributes can be inserted or redefined. A subclass defined by restriction can only introduce new attributes if they are optional, derived or have default values. A superclass instance can be made a member of some subclass using the SUBTYPE operator, but the reverse is not possible.

3.3.1.4 OPERATIONS

Instead of set operations union, difference, intersection and Cartesian product, typical list operations like first, rest and append are provided. The predefined function "setof" removes duplicates from a sequence; while "sort" permits element re-ordering according to any predicate. Other standard functions permit subsequence extraction and desetting. Instead of set membership or inclusion operators, predefined Boolean functions determine if any, all, none, exactly N, less than N, more than N elements of a sequence satisfy a condition. The empty sequence is denoted []:type, where type is the base type of the sequence. An example:

[]: SEQ (Name: string AND Age: num)

Two sequences are equal if they have identical cardinality and base type, and their elements are pairwise equal in the correct order. Aggregate functions sum, average etc. are available for sequences of numbers. Additional operators permit subsequence extraction, iteration and desetting. The notation is shown below:

Subsequence: ALL a_seq WITH predicate
Iteration: FOR a_seq WITH predicate DO expression
Desetting: GET a_seq WITH condition

Failures can be generated by means of the FAILWITH construct. Expression1 IF_FAILS expression2 causes expression1 to be undone and expression2 performed instead in the event of a failure. Failure handling can be tailored to individual assertions using CASE_FAILS, which names the assertion associated with each action.

3.3.1.5 PERSISTENCE

Environments are introduced as a way of partitioning the name space, for specifying persistence and for the provision of views. An environment is a denotable value introduced into the language as a modularisation mechanism, and all expressions are evaluated in the context of an environment. There are operators for extending and

limiting an environment with individual bindings and whole environments, in a number of different ways.

There is a global environment in which all values persist. Environments can be employed to structure this global space, since an environment can be shared by any number of environments which extend it in different ways. User programs may contain class definitions if temporary classes must be kept while running an application. A view of an environment E can be defined as an environment defined on E with some bindings DROPped, some virtual bindings added, etc. As the language is expression-based, every expression is an atomic transaction. Compound transactions comprising several expressions can be defined by enclosing them with Transaction and End_Transaction, and these can be nested. The only type evolution currently available is the ability to introduce new subclasses.

3.3.1.6 COMPARISON WITH P-PASCAL

Whereas P-Pascal extends a programming language with orthogonal persistence, Galileo incorporates a semantic data model into the language ML. The semantic data model concept of classification is realised in both languages through name equivalence. Aggregation in the form of automatic object sharing is provided by Galileo, but requires explicit pointers in P-Pascal. Mappings like Address OF Supplier OF PartUsed are employed in place of pointer dereferencing. The exception handling, first class functions and data abstraction mechanisms of Galileo have no equivalent in P-Pascal. Generalisation is a central feature of the language, but requires some further work. For example, an instance of a subclass cannot be removed from this and returned to the superclass (eg. an object cannot change from a Student to an ordinary Person) [Alba89b]. There are several mechanisms for constraint enforcement; but assertions are only checked at instance creation time [Schm90].

Both languages provide bulk data types and operators. Since Galileo classes are sequences, duplicate but distinct elements can arise. Containment, join, intersection, difference and union operators are not provided in Galileo; typical sequence operators like *first* and *rest* exist instead. As in P-Pascal and DBPL, the bulk type is not limited to first normal form records, and thus does not suffer from the disadvantages identified by [Kent]: a designer is not forced to use multiple record to represent a single entity, variants can be employed to represent different kinds of entity by a single entity type, records like

Employee-Skill which do not clearly represent a particular entity can be avoided, composite domains can be supported, etc. In addition, P-Pascal and Galileo allow pointers to be used as references in place of keys, avoiding the problems arising from alternative keys for an entity and how to map between these, entities having no value for a key (eg. some books have no ISBN number), distinction between weak entities, etc. [Kent].

Galileo does not have the simplicity of P-Pascal. In particular, a distinction has not been made between types and extents (classes). This affects the clarity of the language, and introduces unfortunate restrictions. For example, it is not possible to have two classes over the same abstract data type unless one is a subset of the other; once a type has been declared without being defined as a class, its extent cannot be maintained (or queried); on the other hand a type cannot have individual occurrences separate from its extent. Any number of P-Pascal sets of the same type can co-exist. Furthermore, some can be transient and others persistent. Reachability ensures that P-Pascal databases do not contain dangling references; Galileo will remove an instance of a class independently of its references in other objects.

Environments are convenient for dynamic binding, structuring a database and managing the name space. Much of the syntax, particularly with regard to environment operators, appears cumbersome and it seems as if a simpler set of operators could suffice. Galileo is a rich language; however several features described in [Alba85a] have not been implemented [Alba85c]. These are: optional and default attributes, partition and restriction subclasses, multiple inheritance and environments.

3.4 DBPL DESIGN CRITERIA

Design criteria for database programming languages and persistent languages can be broadly classified into five categories: fundamental principles, type systems, data semantics, operations and persistence. A framework for comparing type systems in particular can be found in [Alba89a] and P-Pascal is described accordingly in Appendix B. For an understanding of types the reader is referred to [Card85b, Ohor].

3.4.1 BASIC REQUIREMENTS

The fundamental design principles to which any persistent language should adhere are:

- uniform treatment of data independent of its persistence
- all data objects allowed the full range of persistence
- data type completeness
- a minimum of constructs and uniform rules
- strong and eager type-checking.

3.4.2 TYPE CONSTRUCTORS

The type constructors available in persistent languages vary considerably. These arise primarily from two sources: databases and programming languages. The former discipline has bulk data types and associative structures; and a range of abstractions, from the very limited eg. normalised records, to the very powerful eg. aggregation [Smit77]. Programming languages typically have arrays, records, variants or unions, and later encapsulation, polymorphism and first class functions appeared. The type constructors determine the ease of use and flexibility of the language, for example its mechanisms for handling many-to-many relationships between objects.

Several other issues relate to type systems. The need to distinguish between type and extent remains an open question, and languages differ in their choice of name or structural equivalence - with many using a combination of both. Some designers advocate purely static type-checking, others a mixture of static and dynamic. There is also variation in the incorporation of bulk data. While this is missing altogether from some languages, it is incorporated in others using different types (sets, sequences or bags) and with differing characteristics.

3.4.3 SEMANTICS

Type systems vary in the amount of semantic information they capture and the precision of data description they provide. Constructs for classification, generalisation (subtyping), exception handling and constraint specification vary from language to language, if present at all. Constraint specification can range from the association of a constancy flag with an object, to the definition of complex assertions and the identification of optional,

derived and default values. Protection of data, through type-checking and access rights, is important and entity integrity and referential integrity [Date] should be guaranteed.

3.4.4 OPERATIONS

The languages described above differ in the kinds of operators they provide for bulk data types. Those encountered are: restriction (or selection), projection, cartesian product, natural join, containment, equality, membership test, intersection, union, difference, desetting, ordering and iteration. In many cases the update facility is limited to the alteration of non-key values. Assignment in some database programming languages has pointers semantics, and in others causes replication.

3.4.5 PERSISTENCE

The mechanism of database identification is important for ease of use, and it should be possible to run a program against any collection of databases to be determined at runtime [Atki87a]. The naming system is sometimes lost when data is stored on a database, and there are languages where disambiguating names can cause severe problems. Facilities for handling distribution, concurrency, transactions and data evolution are clearly desirable in persistent languages.

CHAPTER 4 THE P-PASCAL ABSTRACT MACHINE

The P-Pascal runtime system comprises an interpreter that includes routines for accessing the persistent store. These form a sub-system known as the PPOMS (Persistent Pascal Object Management System) which is an extension of the CPOMS of [Brow85]. This chapter presents an overview of the P-Pascal abstract machine, before describing the CPOMS persistence mechanism. This is necessary to establish the persistent environment within with the data-oriented extensions have been introduced. These take the form of object clustering, metadata management, security enforcement and bulk data constructs, and are presented in chapters 5 to 8. The last section of this chapter describes how the interpreter was tailored to accommodate the constraints imposed by the persistence environment.

4.1 THE INTERPRETER

The P-Pascal interpreter is based on Wirth's P-Code machine [Wirt71]. As such it expects object code input in the form of a sequential file, rather than a graph on the persistent store, as is done in PAIL [Dear87]. PAIL graphs were designed to provide a binding mechanism between a source program and its encoding, to support first class procedures in particular. The graphs retain information gathered by the compiler, which permit optimisations and can be used by utilities such as diagnostic aids and syntax directed editors [Dear88]. The conventional approach of sequential code files appears better suited to the Pascal abstract machine, and more efficient; it also does not require the Persistent Object Store to have sufficient space for storing object programs.

Wirth's Pascal machine architecture was adapted to accommodate the persistence mechanism. The abstract machine was constructed using a layered approach, in which successively more complex data objects are supported at each level.

The Object Layer: Data objects are merely seen as chunks of bits;

The Type Layer: Objects are associated with some type, and clustered and protected accordingly (where a type has a name but no structural properties);

The Structure Layer: Objects can contain components, and be organised into records, arrays and sets; type information reflects this structuring, and security on record components is provided;

The Semantic Layer: Objects and types can be associated with constraints, derivations, inheritance hierarchies, etc. This layer has not been implemented.

The CPOMS and its extension are described in the next section, to establish the constraints this imposes on the rest of the interpreter. Thereafter, relevant aspects of the abstract machine are discussed with reference to these constraints.

4.2 THE CPOMS

The CPOMS persistence mechanism, database organisation and database operations are described below. For further information, the reader is referred to [Brow85] and [Brow89].

The persistence mechanism is based on reachability as the criterion for determining the data to retain on disk. This involves the explicit storage of certain "primary" objects on a database. The system automatically writes to disk all objects referenced from primaries, by recursive pointer traversal. For this reason pointer data precedes other values in every object, and each object has a header giving, inter alia, its size and the number of pointers it contains. All persistent data are heap objects; they migrate between the persistent and transient heaps.

In the CPOMS, primary objects are stored in special data structures termed tables [Carr]. In the P-Pascal system, the table structure is replaced by a set. The root of persistence in each database points to a single set termed the Primary Set, comprising all objects that have been explicitly Saved in that database. The elements of this set contain the object name and location, as well as security references. These object pointers are untyped; but this is only an internal structure which is not visible in Save, Drop or Locate.

4.2.1 OBJECT HEADERS

PS-Algol headers were designed to suit the operation of its abstract machine. As a result, each type had a different header format. In the PPOMS, with its layered approach, all object headers have identical structure and resemble more closely those used by the Napier Store [Brow89].

Every PPOMS object has a four-word header. The first word comprises:
bit 0: is_primary flag (set if this is a primary object)
bit 1: changed_flag (set when object is altered on the heap)
bit 2: reachable_flag (used by the garbage collector)
bit 3: read_only flag (set if object cannot be written to)
bit 4: part_hidden flag (set if some components are read-protected)
bit 5: part_protected flag (set if some cannot be written)
bit 6: commit_flag (set if object has been commited to the store)
bits 7-31: number of sets (required by the interpreter)
The second word gives the size of the object, excluding its header, and the third stores the number of pointers in the object. The last word of the header points to the type of the object.

4.2.2 THE PERSISTENT STORE

The CPOMS divides the store into a number of databases, each of which can be accessed by one writer or many readers. The databases are not totally separate entities however, in that an object can point to data in any other database on the store. When a program opens a database, it specifies whether this will be used in read or write mode. The system automatically opens all other databases referenced from there in read mode. Any database can be re-opened by a program to change its mode.

4.2.3 ADDRESSING

All databases share the same address space. In order to make the subdivision of the store into databases as flexible as possible, the address space is divided into a number of partitions, each 65K in size, and individual partitions can be allocated to databases at will. A partitions file records the database to which each partition has been allocated. Indirection is used to access the persistent store, in order to provide for object-level addressing and to simplify garbage collection. Each database comprises two files: an Indexfile and a Datafile. A persistent address, or PID (Persistent IDentifier), corresponds to a single word in the Indexfile. This word contains the Datafile address of the object. A PID comprises a sign-bit, a partition number and an offset, as shown in figure 4.1. Since partitions are allocated consecutively, the partitions file is used to translate a partition number into an indexfile location: the start of this partition in the indexfile. By adding the offset to this value, the exact word in the database's indexfile is obtained. The

mapping from a partition number to a specific database and indexfile location is stored in memory, to accelerate address translation.

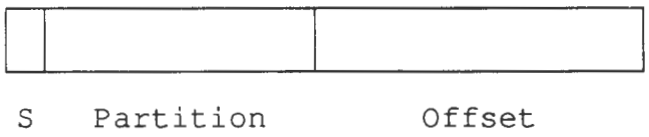


Figure 4.1 A persistent address or PID.

A persistent store address occupies one 32-bit word, and has its most significant bit set in order to distinguish it from memory locations. When an object is retrieved from the persistent store and placed on the local heap, an entry is made in a local data structure called the PIDLAM (PID to Local Address Map). This records the PID:LON association indicating that the object with this PID has been placed at memory location "LON" (Local Object Number). This ensures subsequent accesses to this object do not consult the store. When dereferencing a PID, it is overwritten by the LON where the target object was copied. In this way subsequent use of this pointer will have no address translation overhead. In addition to these two items in each entry, the Datafile location of the object is also stored in the PIDLAM to accelerate writing objects back to the persistent store. Objects are only retrieved from the store if their data is directly needed by the program. On fetching an object, any pointer values remain as PIDs until explicitly dereferenced; a PID is translated to a LON before it is stacked.

4.2.4 DATABASE OPERATIONS

All persistence operations have been implemented in such a way as to permit automatic error recovery. This enables an error record to be returned in the event of failure, without causing program termination.

4.2.4.1 DATABASE CREATION

A Database Directory File records information on all databases in the store. This is the first structure affected by a CreateDB; it is extended to include an entry for the new database (its name, passwords, etc). At the same time, an initial partition is allocated to the new database and recorded in this Directory. An indexfile and datafile are created for

the database, and initialised to contain its root object, with information on its file sizes, its allocated partition and the fact that there are as yet no other databases reachable from here. An OpenDb is then executed, so that the database may be used immediately.

4.2.4.2 OPENING A DATABASE

On opening a database, the system first checks existing (read-/write-) locks on that database. If this indicates the desired mode of opening is possible, the program's lock on that database is recorded. This is achieved with the aid of a single file which controls access to all databases on the store, namely the Database Directory File. The first object retrieved is the database's root object, which is stored at a fixed address. It contains information enabling the database to be addressed and extended, a pointer to the primary set and a list of "reachable" databases. These structures are copied into memory to support the external addressing mechanism. The reachable databases are opened for reading by means of recursive calls to OpenDb.

4.2.4.3 COMMIT

The Commit routine is responsible for updating the persistent store. Its tasks are to write back database objects which have been changed and record all newly persistent objects. Objects retrieved from the store will have an entry in the PIDLAM; hence this structure drives the Commit routine. All objects have changed-flags, so that altered objects are easily distinguishable and unnecessary writes are avoided. Newly persistent objects are those which have become reachable from another database object. Thus these can be identified by examining the pointer fields of objects referenced in the PIDLAM.

Commit first allocates addresses to new objects, at the same time checking that no changed object originates from a read-only database (this aborts the Commit) and copying their original values to a Before Looks File. Thereafter, all new and changed objects are written to the persistent store and their change-markers are reset. If the storage of an object causes a new database to be reachable from that on which its "parent" is stored, this information is recorded. Before an object is written back to the store, all its pointer fields that are LONs are translated into the corresponding PID values in the output buffer. In this way the LON is retained in memory for future use, while ensuring that only persistent addresses exist on the store.

4.2.5 PERSISTENT STORE GARBAGE COLLECTION

When the persistent store is not in use, a separate Disk Garbage Compaction Utility can be run. The PPOMS Utility is an adaptation of the garbage collector of [Camp] which processes the data in two stages. First the transitive closure of every primary set in every database is found; in so doing, every reachable object is marked in its Datafile. The second step compacts a database by traversing its Indexfile and creating two new files. The original Datafile is consulted for each entry in the original Indexfile; if the object is marked, it is written to the new Datafile and its address is written to the new Indexfile. Otherwise, the Indexfile entry is marked as free.

4.3 INTERNAL REPRESENTATION OF OBJECTS

This section explains the constraints imposed by the CPOMS on the interpreter, and outlines how PS-Algol deals with these. An alternative approach has been employed in the P-Pascal machine. The motivation for this is given, after which the incorporation of complex objects is described and a comparison is made between the two systems.

4.3.1 CPOMS CONSTRAINTS

The persistence mechanism imposes constraints on object formats. To facilitate garbage collection of the persistent store and the local heap, pointers must precede other values in all objects, and the size of the object and the number of pointers it contains must be known. This information is stored in object headers so that everything necessary to determine reachable data can be obtained directly from the object.

Thus P-Pascal differs from Standard Pascal in that heap garbage collection is based on reachability; pointers must be kept separate from other components and object headers exist. It differs from PS-Algol in that pointers are typed and can point to data of any kind, including simple values. The language has copy semantics for structured object assignment. Other differences which are relevant here: the heap is transparent to PS-Algol programmers; and data types are divided into two kinds: those that exist only on the heap and those that exist only on the stack. In P-Pascal any kind of object can exist on both stack and heap. Before discussing how P-Pascal data can be adjusted to suit CPOMS constraints, the approach taken in PS-Algol is sketched.

4.3.2 THE PS-ALGOL APPROACH

This language complies with the CPOMS constraints by representing strings, vectors, structures and tables by means of pointers to heap objects; and attaching a header to all heap objects. Scalars (integers, reals, characters, booleans, pixels) are treated differently, in that such variables exist only on the interpreter's stack and do not incorporate a header. Such data cannot be made to persist without being "wrapped" in a structure.

4.3.3 THE P-PASCAL APPROACH

4.3.3.1 OBJECT FORMATS

Every object comprises a header followed by the data of the object, with pointers preceding non-pointer values. The only additional information is the read-flags indicating component security. There is one bit for each word of the structure, to indicate if the corresponding word may be accessed. These flags will be discussed further in the chapter on security.

To permit heap garbage collection, it is essential that the pointers in every variable be determined. This requires that total size and number-of-pointers be associated with every variable. P-Pascal variables are thus also created with headers; besides permitting garbage collection, this ensures that objects of the same type have identical structure, whether they be on the stack or the heap. An alternative approach is to represent all variables as pointers to heap objects; but this is clearly a less efficient solution.

A P-Pascal activation record comprises parameters, a context part and variables. The context part contains a static link, a dynamic link and a return address [Davi81] as well as security-related information and a counter word. The number of parameters and the number of local variables is kept in the half-words of the counter. These values give the base for the algorithm that marks reachable data. On procedure entry, a VarDecl instruction is generated for each local variable. This gives its size and the number of sets and pointers it contains, enabling a suitable header to be created. The other parts of the header (such as the type-pointer) are zeroed, as they are not required for objects not on the heap, only for objects which can persist. Value parameters are handled in a similar way. The scoping of security information is described in chapter 7.

4.3.3.2 NESTED STRUCTURES

One of the objectives of the P-Pascal implementation was to consider how easily the CPOMS persistence mechanism could support different data representations. As a result, the Standard Pascal method of embedding (or nesting) structured data is employed. Sets with large domains (i.e. any base type except BOOLEAN, CHAR or enumerated type) are not represented internally as a single object, since they serve as the bulk data type. They are represented as three pointer words known as a "setrec"; their organisation is discussed in detail in chapter 8. For the moment we note simply that these need to be distinguished from other pointers in an object, as they have a copy semantics on assignment, as well as a different equality semantics. That is, if *ptr* is a pointer in an object, then only that value is used in assignments and tests; if it is part of a setrec, the data it references must instead be assigned or compared for equality. For this reason, all headers include a number-of-sets count, and setrecs are separated from other components by the interpreter. The distinction between pointers and setrecs is transparent to the PPOMS, and the number-of-pointers in a header includes setrec pointers and conventional pointers.

In contrast to the PS-Algol and Napier88 abstract machines however [PAM, Morr90], records and arrays are nested within other structures, instead of using pointers to separate objects. This implementation decision was taken for three reasons. Firstly, it conforms to the Standard Pascal representation. Secondly, it can be used to investigate the ease with which such complex objects can be handled by the CPOMS architecture. Thirdly, reduction in the number of pointers decreases access and storage times. In very large databases, the resulting reduction in the number of objects enables more data to be accessible using a 32-bit address. The tradeoff lies in the extra space occupied by persistent data on the local heap, in the case of components which are not actually used by the program. The complexity introduced into the compiler as a result of the separation of nested structures into pointer and non-pointer values is not significant. The main reason for nesting objects directly was to investigate the effect of such data organisation in a CPOMS environment.

4.3.3.3 CANONICAL FORM

P-Pascal type declarations are translated internally into canonical form. The algorithm classifies objects as pointer, setrec, non-pointer or mixed. Any type definition is re-

organised so that pointer data precedes non-pointer. In the pointer section, setrecs precede other pointer values. To achieve this organisation, mixed objects are seen as three separate entities, containing the setrec, pointer and non-pointer information respectively.

The types classified as non-pointer are integers, reals, booleans, characters, scalars and subrange types, sets of small base type (i.e. char, boolean or enumerated type), and those (variant) record and array types having only non-pointer components. Pointer data comprises pointer types, and record and array types having only pointer components. Setrecs arise only when sets of large base types exist. The data types which are "mixed" are those array and (variant) record types having components of more than one kind.

The internal representation of pointer and non-pointer types is identical to Standard Pascal, except that a header is incorporated in every stack and heap object. In addition the fields of a record are re-ordered alphabetically. This is transparent to the programmer, to whom declared field order is significant. A mixed type is represented in three parts; the first comprises all setrec components, the next all the pointers and the last contains all the non-pointer values. Where such objects are nested inside others, the parts can be disjoint. As a consequence, a reference to a component of mixed type is translated into two or three separate references by the compiler. Eg. X.REC1 := Y.REC1; is coded as three assignments if all three kinds of field exist: assigning firstly the setrecs, then the pointer-values and finally the non-pointer values.

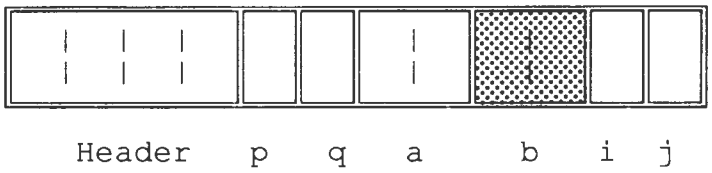
The fields of a mixed record are divided into setrec, pointer and non-pointer fields; and within each of these sections, the fields are ordered alphabetically. If a field is an embedded record or array, it will occur in one or more sections, depending whether its fields are all pointers, all non-pointers, all setrecs, or mixed. The fields of variant records are re-arranged into setrec, pointer and non-pointer sections, each having fixed fields preceding variant data, and fields are ordered alphabetically within this framework. Thus variants internally comprise fixed setrecs, variant setrecs, fixed pointer values, variant pointer values, fixed non-pointer data and variant non-pointer data, in this order. All variant sections are allocated sufficient space to accommodate the case with the largest demands of that kind.

The algorithm for translating a mixed object X into its internal representation proceeds as follows. If X is a mixed record type its mixed components are translated first, if any. If

The algorithm for translating a mixed object X into its internal representation proceeds as follows. If X is a mixed record type its mixed components are translated first, if any. If X is a mixed record type comprising fields that are setrecs, pointers and non-pointers, it is replaced by three separate sections - containing all setrec fields, all pointer fields and all non-pointer fields respectively. If X is an array of mixed records of type M, M is translated first and then X is replaced by three arrays, comprising setrec components, pointer components and non-pointer components respectively. If X is a structure comprising data of two kinds, its components are subdivided in two in a similar way. Figures 4.2 and 4.3 give examples of P-Pascal type declarations and their internal representation.

Example 1: Field b is shaded below.

```
TYPE R = RECORD b:real; i,j:integer; p,q:aptr; a:real
         END;
```



Example 2: Field mx is shaded below.

```
TYPE R =    RECORD b:real; i,j:integer; p,q:aptr; a:real
            END;
     M =    RECORD e,d : real; mx : R; p1,p2 : someptr
            END;
```

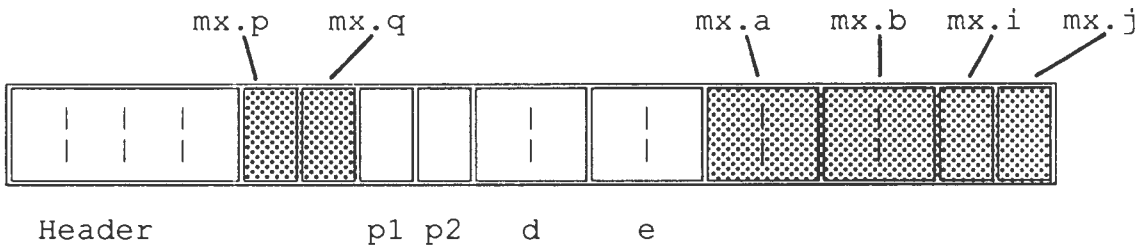
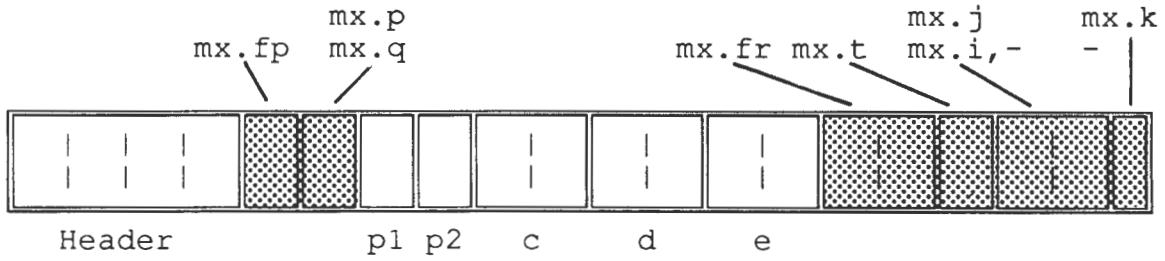


Figure 4.2 Internal Representation of mixed structures.

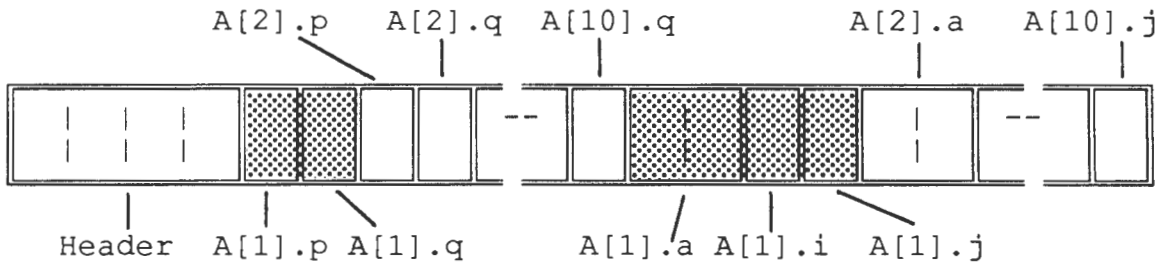
Example 3: Field mx is shaded below.

```
TYPE REC = RECORD fp : fixedptr; fr : REAL;
               CASE t : BOOLEAN OF
                      TRUE : (i : INTEGER; p : someptr);
                      FALSE : (j,k : REAL; q : anotherptr)
                END;
   M =RECORD c,e,d : real; mx : REC; p1,p2 : someptr
               END;
```



Example 4: The first array element is shaded below.

```
TYPE R = RECORD i,j:integer; p,q:aptr; a:real
       END;
       A = ARRAY [ 1..10 ] OF R;
```



Example 5: Field z[1,2] is shaded below

```
TYPE REC = RECORD i : INTEGER; p : someptr END;
       A = ARRAY [1..5, 1..5] OF REC;
       R =    RECORD x : REAL; y : SOMEPTR; z : A;
                     s : SET OF someptr
              END;
```

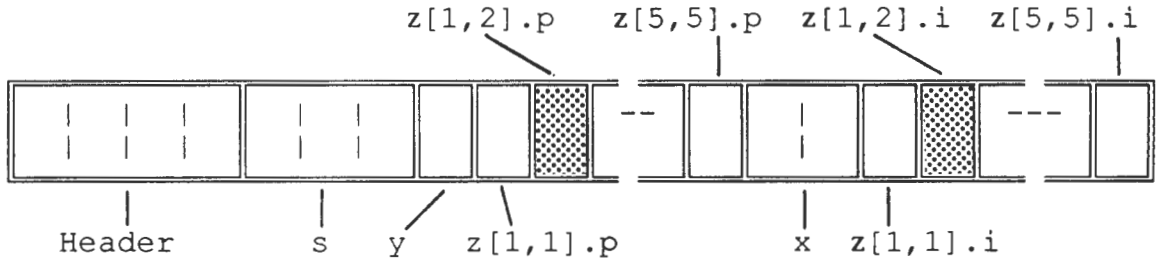


Figure 4.3. Internal representation of structured objects: further examples.

4.3.3.4 COMPONENTS AS PARAMETERS

A potential problem arises when a nested object is passed as actual argument to a VAR parameter. Firstly, embedded objects do not have headers; secondly, if the object is of mixed type, it has not one address, but two or three. Inclusion of headers with all embedded objects is possible; a one-word header-pointer was designed and its use as a prefix to data of all types was investigated. The compiler replaced parameters of mixed type by two or three parameters, so that setrec, pointer and non-pointer sections could be passed separately. The presence of these headers greatly complicated the creation and manipulation of objects, since most operations had to detect the presence of headers and handle them differently from data.

VAR parameters are therefore implemented using call by value return where call by reference is not possible. This enables the system to construct a header for embedded objects, and combine setrec, pointer and non-pointer sections where necessary, before passing control to the routine. This has the obvious disadvantage of being expensive in terms of space. However, this is necessary only when passing components instead of entire variables. In addition, when a bulk data object occurs as VAR parameter, only the three setrec words have to be copied.

4.3.4 COMPARISON

The representation of embedded structures by means of pointers to separate objects could be adapted for P-Pascal; it offers several advantages but also some disadvantages over that described above. P-Pascal permits pointers to data of any kind, including simple types. Thus if an object is made to persist and it has a pointer to (say) an integer, the integer must become a separate database object in its own right. As such it requires a header. It is impossible to dispense with a header altogether on the persistent store: every object must at least have one bit to indicate header-is-absent, as well as a mark bit for garbage collection To reserve space for this within the object itself reduces the bits available for storing an integer value to 30, and requires software checking for overflow. Even more header information is required on reals (to indicate they occupy two words), pointers (their type) and strings (their type and size). The alternative of disallowing pointers to simple types was not considered, as P-Pascal aims at extending Standard Pascal, not restricting it. It can be useful in certain cases to store simple values directly on a database, and as a result the language provides truly orthogonal persistence.

The PS-Algol approach to embedded data could be extended by representing all components by pointers to heap objects, even simple ones. This would result in two kinds of object: simple objects, comprising a header and a single value; and structured objects, comprising a header and a sequence of pointers. This in turn would mean that the problems associated with the division of structured objects into setrec, pointer and non-pointer sections disappear, and the reachability algorithm is extremely simple. However far too many pointers result, with corresponding increases in access times and space utilisation.

PS-Algol distinguishes simple objects from structured objects in several ways; in particular, simple objects have copy semantics for assignment but structured data does not. In Pascal however, if a component is structured all its values must be copied on assignment. Thus if it were represented as a pointer, it would have to be distinguished from program-declared pointers which have different semantics for assignment and equality.

The method of using pointers in place of nested objects has the following advantages:

- the amount of persistent store data retrieved into local memory is kept to a minimum, since component objects are not fetched until explicitly requested;
- call by reference is always possible;
- variant records waste less space, in that the difference in variant sizes is less likely to be great.

The advantages of directly embedding objects are:

- a complex object occupies less space, since the pointer and the header words are avoided at every level of nesting;
- far fewer pointer traversals are required to process data - which is particularly significant in the context of the persistent store, where each generally involves two disk accesses;
- sets can be processed more efficiently when all the information on an element is kept together, as shall be seen later;
- a language with a copy semantics for assignments does not lend itself to indirect representation of values.

Another difference between the two approaches lies in the fact that P-Pascal has fewer but larger objects, rather than many small objects. The effects of storing large objects in a persistent environment requires further investigation.

4.3.5 OTHER PPOMS EFFECTS ON THE INTERPRETER

The interpreter has to be adjusted to accommodate heap garbage collection, which is different from that of Standard Pascal. Rather than freeing up space as indicated by the program (through a Dispose), the system discards objects which are unreachable as well as objects that exist on the persistent store. The present system does not discard persistent objects if they have been altered since the last Commit. They cannot be written back to the store, since atomicity would then be forfeited. A checkpointing mechanism [Brow89] or the use of a temporary store for objects swopped out of the local heap is required; however this has not yet been implemented. With the current version of the system, the onus is on the programmer to ensure that Commit is used appropriately, to enable objects to be written back to store as often as appropriate.

As shown in the previous section, allowance is made for heap compaction by introducing object headers and separating pointer and non-pointer components. In every scope, in addition to the main stack, a separate stack is used for temporaries of type set, since these can contain pointers to temporary heap objects which must be retained by the garbage collector. The alternative requires associating a bit with every word, indicating if it is a pointer or not [Davi81].

4.4 CONCLUSION

The PPOMS is an extension of the CPOMS that permits type clustering, metadata maintenance and security enforcement. As sets are the P-Pascal bulk data type, a database root points to a set rather than a table; but is functionally identical to the CPOMS root table. Object headers resemble more closely those used in the Napier implementation [Conn89], in accordance with the guidelines laid down by the PISA project [Atki87c, Atki88]. In all other respects, the PPOMS is identical to the CPOMS [Brow85] - database representation, supporting file structures and internal and external addressing mechanisms are identical, as are the algorithms implementing database creation, opening and Commit.

The remainder of the P-Pascal interpreter differs in several ways from its PS-Algol counterpart. All objects include headers, and data of any kind can persist without having to be wrapped in a structure or vector. The location of data on heap or stack is not determined by its type. Hence even variables have headers, and activation records are no longer as simple as in Standard Pascal. While retaining the separation of pointer and non-pointer values, complex objects have been implemented as nested structures, to investigate the potential of the CPOMS for supporting such entities. The results showed that this representation does not create any major problems, except that embedded objects require call by value or value return. Moreover, several advantages to the nesting approach were identified. In contrast, the use of pointers to bulk data objects required distinguishing these from other pointers because of their different assignment and equality semantics. Additional implications of embedding objects will be highlighted in the chapters that follow, when clustering, metadata maintenance, security enforcement and bulk data handling are discussed.

CHAPTER 5 CLUSTERING

The object placement strategy used by the CPOMS has been modified in the PPOMS by the introduction of a type-sensitive, family-clustering storage mechanism. It aims to keep together physically that which is likely to be accessed together and to facilitate extent processing by a future Online Interface Utility. This chapter presents the object placement strategies of the POMS [POMS, Cock] and CPOMS [Brow85], followed by some alternatives considered for the PPOMS. The adopted strategy and its implementation are then described.

5.1 OVERVIEW

Most clustering schemes are based on the principle that objects should be clustered with the data they reference [Benz90a, Benz90b, Chan89, Schk, Shan, Stam, Yu]. That is, if X has a pointer to Y, both should be stored in the same block. In the CPOMS X is stored near Y - that is, directly after its youngest new descendant - only if both X and Y are new objects. If only one is new to the store, space will not have been kept nearby for the other. In the POMS [POMS], its predecessor, objects were categorised into classes. Each logical block contained members of only one class. Strings and vectors were allocated to these "fractions" according to their size. A list of Freed PIDs and another of used logical blocks was maintained for every class. The former was used in address allocation, the later served as the extent of the class. Some of these ideas regarding type-clustered organisation have been incorporated in PPOMS clustering; the strategy has been extended and the representation of placement data has been altered.

The aim of the PPOMS is to place related objects together. If a new object cannot be placed near any existing relative, it is located in such a way that parents, siblings or children subsequently created can be placed nearby; the system prevents too many un-related objects being placed on the same block. An extent is the collection of all occurences of a specific type. Where this is easy to obtain, there are several advantages: determining if a type is empty; efficiently manipulating the type as a whole; facilitating type constraint enforcement, etc. Clustering is an optional property of P-Pascal databases, enabling the designer to save space instead if this is more suitable for a particular application environment. A CreateDb parameter indicates whether or not the new database is to be clustered. A program can use both clustered and un-clustered databases, and can inter-link these where necessary.

5.2 PLACEMENT STRATEGY

The object placement strategy which can be enforced is influenced by the characteristics of the persistent store which differ from conventional database systems, as outlined below. This section presents some alternative mechanisms that were considered and proposes a strategy to suit a persistent system like the CPOMS.

5.2.1 DIFFERENCES FROM RELATIONAL DATABASES

In the CPOMS, all databases share the same address space and the objects in a database share the same files. Each object has an entry on two files: indexfile and datafile. This architecture does not lend itself to clustering groups of related objects. As the distinction between transient or persistent data is transparent, disk placement specifications in type declarations are highly undesirable. A database can include objects which are no longer reachable; these are purged at the next Disk Garbage Collection, so clustering must be re-applied on compaction.

A database is a collection of heterogeneous, inter-linked objects of different types. Data structuring is not uniform as in relational databases, network database ("sets" only) or hierarchical databases (trees only). In the PPOMS nested structures are not represented by means of pointers, so there are far fewer objects to place on the store, fewer objects related to any given object, and larger objects are common. In particular, the proportion of references which point to an object of the same type should be fairly high.

5.2.2 CANDIDATE STRATEGIES

Several candidate object placement strategies were investigated, ranging from a simple scheme keeping objects of the same type together, to complex ones where programmers specify exactly how to divide up the database amongst different types: eg. for every type state which blocks it uses and the number of occurrences per block or group of blocks [DMS]. A strategy could aim at placing a newly persistent object X on the same block as a related object. Different kinds of relationships can be targeted:

- the "parent" object causing X to be commited (the first persistent object in which a pointer to X was encountered);
- any "child" object (any Y such that X contains a pointer to Y);

. any "child" object of a specific type;
. the specific "child" object referenced by a particular pointer within X.

Different schemes are possible depending on the relationships involved and the priority assigned to each. As an extreme case, the placement strategy could be: place X with its parent, else with a specific child, else with some child of a given type, else with any child, else with other instances of its type. If a declaration specifies an object must be clustered with its "children", child selection rules can be developed. The choice could depend on the amount of free space in the child's block, or on the characteristics of the data structure.

Space could be reserved on a disk block for the future descendants of its occupants. The database designer would specify the percentage, which would then apply to all blocks. This can be incorporated with any of the above strategies. More detailed information can then be given: for example, individual percentages can be specified for different data types.

5.2.3 ALTERNATIVE SPECIFICATIONS

The complexity of type declaration increases with the flexibility of the placement strategy, and the database orientation becomes more evident. Unless clustering is applicable to all types on a database, clustered types must be distinguished by a declaration such as

TYPE t = ... CLUSTERED;

To cluster objects of different types together without a fixed strategy for doing so, more cumbersome syntax is required, such as

CLUSTER name OF type1 [percent1],..., typeN [percentN]
TYPE t = ... CLUSTERED component1 [, ..., componentN];

or TYPE t = ... CLUSTERED childtype1 [, ..., childtypeN];

To tailor both indexfile and datafile placement strategies to an application, further syntax is needed. The situation is also aggravated if portions of a "cluster" (disk block) are to be reserved for specific purposes, eg.

DATA CLUSTER cname OF parenttype WITH childtype1 [percent1], ...,
childtypeN [percentN];

5.2.4 STRATEGY ADOPTED

The Placement Strategy chosen was that which provides the most flexibility without burdening programmers with any new terminology or syntax. As the block is the unit of disk access, the strategy is based on this. Although clustering objects on contiguous blocks is possible, this would not save disk head movement in general, since indexfile accesses alternate with datafile retrievals.

Indexfile blocks are reserved for individual types; that is, the occurrences of a specific type T are only stored on the blocks assigned to T, and they are the only occupants of these blocks. This is termed type-clustering. P-Pascal pointers can only reference objects of a named type. Thus every heap object stored on a database is associated with a unique named type, which is used to determine where it may be stored. Each system type is given a name and treated identically to program-defined types, as are the Standard types.

In addition to the association of one type with each block, "family-clustering" attempts to store an object on the same block as others to which it is related. Two objects are said to be related if one contains a pointer to the other. An object is placed on the same block as its parent if this is of the same type, and space is available on that block; otherwise it is placed on the same block as any of its children of that type if possible. Where neither is possible, any block assigned to that type is used. A load factor is associated with a database in a CreateDb. Once this portion of a block has been filled by arbitrary instances of the type, the remainder is reserved for objects related to these occurrences. The combined strategy groups together objects of the same type; and within this, keeps related objects nearby. Thus the system aims primarily at clustering related objects of the same type.

Datafiles are type-clustered, on the assumption that lists of like objects are common data structures. An attempt was made to apply family-clustering to datafile blocks. The address allocation algorithm first looked for a relative of the required type such that there was space on both its indexfile block and its datafile block; otherwise any blocks containing a relative were used if possible. This strategy was abandoned however, primarily because of the amount of space required in keeping information for every datafile block. In addition, disk garbage collection did not preserve family-groupings on datafiles. Since these files are compacted, instances of a type remain in the same order

on the file, but block boundaries are altered. Maintaining lists of free datafile space would make address allocation too slow; rebuilding family-clusters in the datafile on garbage collection would require application of the reachability algorithm to a database during the compaction phase.

Either type- or family-clustering could be implemented in isolation. However, without the latter there could be a large number of objects that cannot be stored on the same block as related data. Without type-clustering, there is no placement strategy for datafiles, and extent processing is slower. The existing data placement routines could easily be adapted to support either strategy alone. At present only type-clustering can be employed in isolation, by setting the family-clustering loadfactor to 100 percent.

5.3 REPRESENTATION

The system requirements imposed by the above strategy are discussed, and two methods of storing clustering information are presented. The representation of placement data is important because it is voluminous and must be efficiently accessible if Commit times are not to be appreciably affected.

5.3.1 REQUIREMENTS

For extent processing it is necessary to map a type onto the list of the blocks where its instances are stored. Recording the indexfile blocks assigned to a type is sufficient; and requires considerably less space than a list of datafile blocks.

For address allocation during Commit, the following information is needed:

(A) Given the PID of a relative, is there a free PID on the same block (a "family-PID") ?

(B) Given a type, what is the next PID to allocate to an instance, assuming it cannot be stored with a relative (a "general-PID") ?

(C) Given an indexfile block, what is the percentage of free PIDS on this block (i.e. does it have space for objects unrelated to others on the block)?

(D) Given a type, what is the next datafile location to assign to an occurence ?

More complex placement strategies would require additional knowledge on blocks, such as the types of object that may be stored there and the proportion of space allocated to each.

Two methods of storing object placement information on a database are described below. The initial scheme was implemented and then replaced by the second design which is more compact and more efficient for address allocation. The two designs are outlined below.

5.3.2 INITIAL DESIGN

Files are unsuitable for storing placement data, because information on any block or on any type must be rapidly locatable. This information must therefore be maintained on the persistent store itself. Since placement is type-sensitive, every type object incorporates NextPID and NextLOC fields, to satisfy requirements (B) and (D) respectively. Several alternative representations for the block-related information were considered.

The blocks assigned to a type could be given as a list, a list of arrays, a bit vector, or a set. A list is too expensive in terms of both space and access time, and a list of arrays is almost as inefficient. A boolean array would have to be too large, since the PIDs of a single database are not consecutive numbers. A set permits indexing the information for rapid random retrieval, but is fairly expensive in its space utilisation.

The representation selected initially was that of a set attached to the type object. An element was created for each indexfile block. It contained B - the block number by which to identify the block; N - the number of free PIDs in the block; and F - the first free PID on the block. This information was represented in two words. B was stored in bits 1 - 15 of the first word, and N in bits 16 - 31 of that word. This word served as the key of the set, for rapid location of the data for a given block. Free PIDs were kept in a list starting at F (see figure 5.1).

In the CPOMS, the Free-Indexes-List is a single list of PIDs currently not in use [Camp]. With a clustering strategy this information changes considerably. In particular, there can be many blocks with free entries, and a block can have many empty slots. A separate list of Free-Indexes was built for each block, with the list header in the type object. Each entry contained the PID of the next free slot and the list terminated with -1.

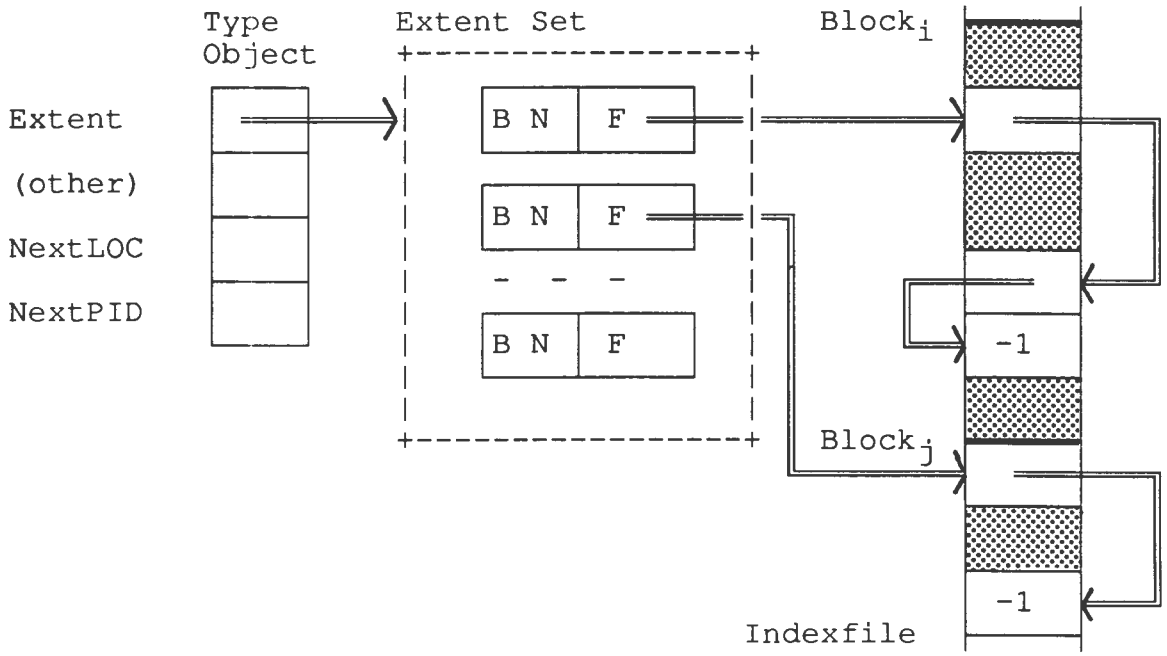


Figure 5.1 Original representation of blocking data. Shaded areas indicate entries containing datafile addresses.

Garbage Collection was complicated in this scheme, since block details changed during compaction, requiring that this information be altered as each indexfile block was completed. The use of sets impacted on space requirements. The design was replaced by a simpler storage of block-descriptions, to diminish the amount of placement data, simplify garbage collection, permit faster address allocation and reduce storage space.

5.3.3 CURRENT DESIGN

By keeping a single list of free-slots per block (together with a tally) general-PID allocation is retarded: each block assigned to a type has to be examined until one with sufficient free slots is found. This organisation was modified to provide a single list of general-PIDs and partition the free slots on a block into two separate lists: one of family-PIDs and the other of general-PIDs. This is illustrated in figure 5.2 and described below.

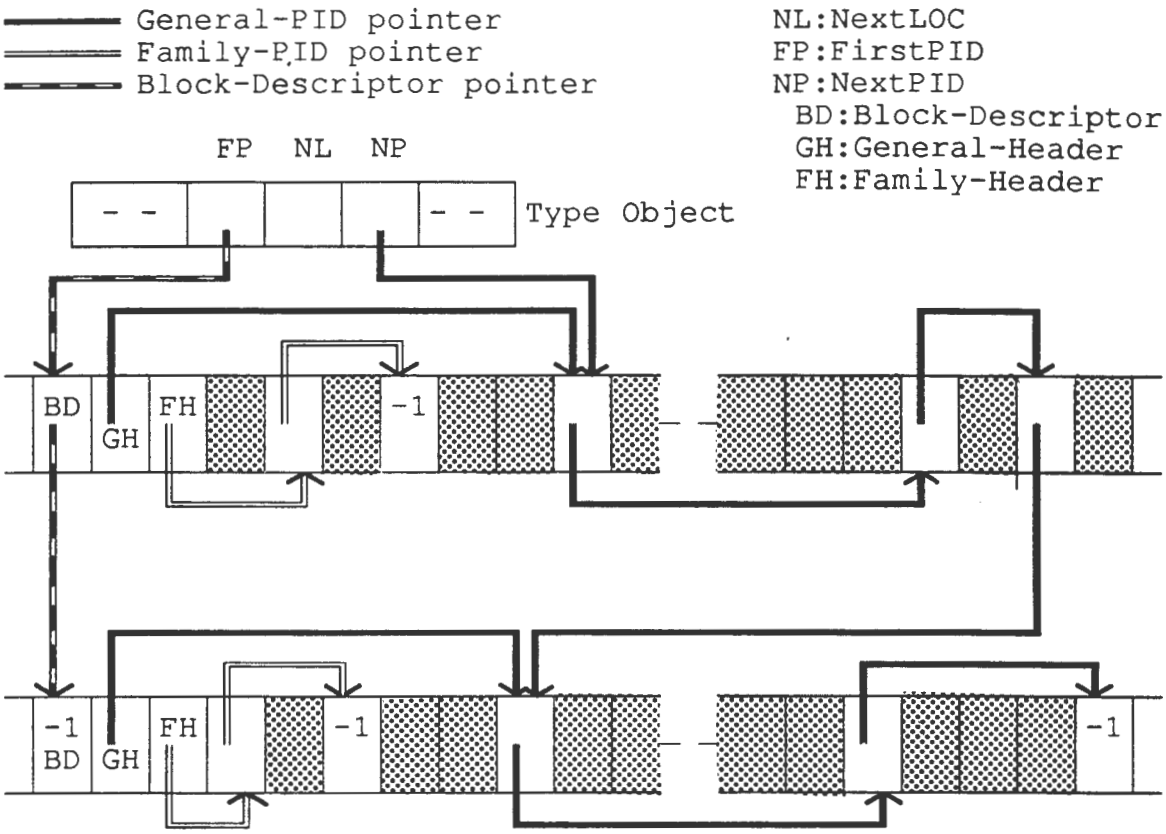


Figure 5.2 Current representation of blocking data. A type with two blocks is shown. Shaded areas indicate PIDs pointing to datafile objects.

Each type is associated with two lists:

. the Block-Descriptor list, representing the extent of the type.

. the General-free list, comprising all free indexfile slots which can be allocated to any object of the type. Family-PIDs are excluded, as these are not available for general use.

Each indexfile block has two lists:

. the Family list, comprising the free PIDs in an indexfile block reserved for "relatives" of its occupants;

. the General-free list, comprising all free indexfile slots which can be allocated to any object of the type (family-PIDs excluded).

The first three PIDs of every indexfile block are reserved for use by the clustering subsystem, and are collectively known as the Block-Descriptor. This contains three values used for extent processing, family-clustering and type-clustering respectively:

. NextBlock : this points to the next Block-Descriptor associated with the type

- Family-Header : this contains the PID of the first entry in the Family-list for this block
- General-free-Header : this holds the PID of the first General-free-list entry in this block.

Placement information for a type is recorded in a type object on every database where that type is used. This comprises FirstPID, NextPID and NextLOC. The extent can be processed by starting at the FirstPID field of the type object and proceeding along the Block-Descriptor list, following NextBlock pointers. The General-free list of a block is part of the General-free list for the type, as shown in figure 5.1. This list starts in the NextPID field of the type object, which contains the PID of the first entry in this list. Each of these indexfile entries contains the PID of its successor; the list ends with -1. Thus this chain proceeds through all the blocks assigned to the type which are not yet filled to their Loadfactor. Each of these entries can also be reached from the General-free-Header in the corresponding Block-Descriptor. Entries in free indexfile slots are translated from file offsets to PIDs during garbage compaction, so that the address allocation stage of Commit need not perform the conversion.

In addition to the lists of free PIDs, a single Blocks-Free-List is kept. This has its header in the database root and terminates with -1; each entry containing the PID of the first entry in the next free block. A block is "free" if all of its slots except the first three (the block-descriptor) are unused. These PIDs are not part of any General-Free list, as they are not reserved for any specific type.

5.3.4 COMPARISON

The current system altered the original representation of blocking data. This information is now separate from type objects, and is instead associated with specially reserved locations on the indexfile. Thus blocks are self-contained, simplifying garbage collection and address allocation. Another major difference lies in the method used to distinguish blocks below their Loadfactor. Instead of keeping a single list of free slots on a block, together with a count, two separate lists are kept. This facilitates the handling of both kinds of PID allocation and simplifies block-descriptor adjustment.

5.4 IMPLEMENTATION

5.4.1 ADDRESS ALLOCATION

When a type object is retrieved from the persistent store for type-checking and security enforcement purposes, its placement information is retained in memory for use by the address allocation routine. Indexfile addresses are allocated as follows. If there is a relative of the new object that is of the same type, and space exists on its indexfile block, a PID is allocated from this block; otherwise the first available PID for this type in the parent's database is used. Once a PID has been assigned, the Block-Descriptor of the corresponding block is updated (its General-Free-Header or its Family-Header, as appropriate). If there is none - i.e. all the type's blocks have reached their Loadfactor limit or the type has not yet been used on the parent's database - a new block is assigned to the type and the new object given a PID from there.

When PID allocation assigns an indexfile slot to an object on the basis of type alone (i.e. it cannot be placed near a related object), the next entry in the type's General-Free-list is used. For family-clustering, the General-free list of the corresponding block is used first; only when this is exhausted are slots taken from the Family list itself. This ensures that slots reserved for related objects are always the last to be used, and no general-PIDs are assigned from there once the block has exceeded its Loadfactor. Whenever a PID is allocated to a new object, the corresponding Block-Descriptor must be updated. This is not an expensive operation as the Block-Descriptor will already be in an indexfile buffer, since it is in the same block.

When a new block is required, the next block in the Free-Blocks-List is used, if any; otherwise the indexfile is extended. Its Block-Descriptor is initialised at the same time, and the object requiring an address is assigned the fourth slot on this indexfile block. The type's FirstPID and NextPID are adjusted to incorporate the new block. If the new block is not taken from the Free-Blocks-List, the type's NextPid is negated so that no attempt is made to update a General-free list when the file is being extended. Adding a new block to the file requires writing temporary values to the remaining slots of the last indexfile block, until the block boundary is reached. These values are written in such a way that each contains the PID of its successor, as expected in free PID lists.

A datafile location is assigned to an object in the database from which its PID was allocated. The type's NextLOC is used and subsequently incremented by the object size.

If this exceeds the block's capacity, NextLOC becomes NIL. If NextLOC was initially NIL, a new block is allotted to the type, and the datafile is extended.

The Commit algorithm performs the following steps for the transitive closure of changed PIDLAM objects (where X is the object under consideration):

1) for each pointer component that is a LON, Commit the referenced object. The PID of X and the LON of its type-pointer are passed as parameters, so that the "child" object may be allocated on the same indexfile block if possible.
2) if X itself is not present in the PIDLAM, allocate a PID to X on the same block as its parent, or any of its children, where these are of the same type as X; otherwise allocate X the next PID associated with objects of this type.

Type clustering necessitates a Commit of every type object having a newly persistent occurrence, as its Nextpid, NextLOC and possibly its Firstpid will have changed. After the address allocation phase of Commit, the clustering data will be up to date. Thus the type objects written to the persistent store in the subsequent step will be correct.

5.4.2 GARBAGE COLLECTION

The reachability phase of garbage collection is unaffected by the introduction of clustering. During the compaction phase, reachable datafile objects overwrite those which are no longer persistent. In a clustered database, this process must ensure that only objects of a single type occupy each block. In the indexfile, Block-Descriptors are altered to reflect the new database organisation and freed PIDs are assigned to free-PID lists according to the loadfactor for the database. The final design for clustering information minimised the amount of placement information attached to a type object. As this information can only be finalised at the end of the compaction phase, it is desirable to retain as little placement data as possible during this process. With the current system, only the NextPID, FirstPID and NextLOC information of the type needs to be kept.

The compaction phase traverses the database's indexfile, creating a new indexfile and a new datafile, and building new lists of free PIDs. Clustering objects by type on the new datafile is simplified because, for each indexfile block processed, the objects it points to are all of the same type. For each indexfile block, free PIDs are allotted to the Family-List according to the loadfactor; any additional free slots contribute to the General-Free-

PIDs list. When an indexfile block is completed, its Block-Descriptor is written to the datafile.

5.5 CONCLUSION

5.5.1 SUMMARY

A method of clustering persistent objects by type has been devised and implemented for both indexfiles and datafiles. In addition, a loadfactor can be set for each indexfile, so that a proportion of every disk block is reserved for objects related to data already stored there. The combination of these ideas is designed to increase the likelihood of objects which are processed together being found together on disk. This chapter described how clustering information is recorded on the persistent store and used by the address allocator. The changes required in the Disk Garbage Collector were also outlined. We conclude with a brief discussion of the effect of PPOMS design decisions on data clustering strategies.

5.5.2 DISCUSSION

The structuring of the store as a single address space, and the mapping of a database onto a single pair of files, makes sophisticated clustering mechanisms like those of conventional database systems difficult to apply. The extent of a type must necessarily require slightly more space and processing time, since this cannot be stored in its own file. The ability to reference objects in other databases decreases the likelihood of keeping the majority of related information together. Since address allocation is not a simple process, any clustering strategy imposed on top of this must perforce require as little work as possible. For this reason, for example, the free indexes are divided into two separate lists according to the type of address allocation (type-clustered or family-clustered). The disk garbage collector requires that an object placement strategy be re-applied with every compaction. In particular, this makes the application of any non-trivial clustering algorithm very costly to apply to datafiles (as opposed to indexfiles, which are not compacted). Even indexfile re-organisation requires considerable adjustment when clustering is introduced, as free slots must be structured to suit the clustering algorithm, etc.

On the other hand, index entries comprise only one word and are stored on a separate file, providing a convenient opportunity for clustering. Each indexfile access brings 1K of 32-bit references into a system buffer. The entries should be organised in such a way that a majority of references are localised and can be obtained directly from the buffer before it is overwritten. In contrast, relational database systems generally keep the equivalent of index entries on the same page as the data to which they point [Ston76,Astr76]. Thus there are considerably fewer per block, making it more difficult to cluster them to suit data usage.

We note that the comments above apply equally to the Napier Stable Store [Brow89], as this also has a single file and address space, a separate index area and a garbage collector; only inter-linked databases have been omitted. Since structured objects can be nested, the store will contain fewer (but larger) data chunks, and a high proportion of references is likely point to data of the same type. Clustering objects with related information of the same type thus appears the best way of keeping together that which is likely to be accessed together.

CHAPTER 6 METADATA MAINTENANCE

P-Pascal metadata is maintained for two reasons: it enables the types on a database to be queried; and it permits dynamic type-checking and security enforcement. This chapter explains how metadata is perceived by the user, how it is represented in the persistent store and how it is handled by the interpreter.

6.1 MOTIVATION

Metadata maintained by the system is used by the interpreter and by users requiring information on existing databases. It can be accessed from within a program or by means of a separate utility. An on-line metadata query facility is useful for new programmers to learn about a persistent store or database, for occasional users to recall type specifications and for regular users to study the mix of data on databases across the store, to obtain cross-reference lists, to check if a name is in use, etc. To obtain this information without automatic metadata provision requires hunting for program listings and checking that these are up to date. Metadata is also needed by Universal Application Programs such as browsers or statistical and diagnostic aids [Owos]. In order for these to access data, some additional language construct such as a dynamically callable compiler [Dear88] or dynamic type declaration [Owos] is required.

6.2 USER VIEW

A global metadata dictionary is maintained for the persistent store, as well as individual directories for its databases. A schema constraining the data that programs may place on the store has been purposefully omitted from the system architecture. This section explains these decisions, and outlines what metadata is available and how it can be accessed.

6.2.1 SCOPE

As P-Pascal uses name equivalence, type names must clearly be unique within any one database. Moreover, it cannot always be determined on which database a persistent object will be placed. Types are constrained to be unique across the entire persistent store, so that a program that has created an object can freely have it stored on any database without conflicts arising.

6.2.2 SCHEMAS

Traditionally, database management systems have associated a schema with a database, which names and describes all possible types that may be used on that database. A subschema selects relevant parts of a schema, possibly modifying type descriptions to suit a particular application's view of the data.

Owoso [Owos] introduced schemas to PS-Algol, along with statements such as "from <database> use <class>" which clearly distinguished transient and persistent types. Acknowledging the importance of complete type definitions within programs, he permitted only integrity constraints to be omitted when a schema type was used in a program: definition sections were thus not appreciably shortened. Although static or dynamic introduction of new types to schemas was supported, problems related to inter-linked databases and schema contention were not addressed [Owos].

A schema facility has been purposefully omitted in P-Pascal, for the following reasons. When persistent languages were proposed, one important feature was the ability of programs to freely add new types of data to the persistent store during execution. Transient and persistent types were indistinguishable. P-Pascal aims at retaining these features; it has no concept of schemas, nor explicit incorporation of persistent types. In a persistent environment there should be no restriction preventing programs from introducing new types; and persistent data should not be treated differently from transient. Metadata maintenance should provide a service by describing types in use; rather than act as a limitation on what can be placed in the persistent store.

With P-Pascal, a programmer will typically run the on-line metadata query facility to check that a type name is unique, before introducing a new type to the persistent store. This avoids the possibility of having to edit, recompile and rerun a program due to the accidental choice of a name already in use. If a program names a type incorrectly, it will not be able to access existing objects of that type nor make any new object of this type reachable from an existing object on the persistent store. A mistake can only go unnoticed if a program creates new primary objects associated with an incorrect type name and at the same time does not deal with existing persistent data of that type. This situation calls for a compiler which can access the persistent store and print warning messages on detecting a new type. As the current version of the compiler does not have

this feature, it is the responsibility of the programmer to check type names before creating primary objects, as is the case when using libraries in other languages.

6.2.3 TYPE-CHECKING

An incorrect type declaration is reported only if and when such an object is Located on, or Committed to, a database. It is clearly inappropriate to automatically raise an error when a program type differs from its database definition, as the program may have no intention of using database objects of this particular type. It is also highly undesirable to have types declared persistent.

6.2.4 CURRENCY

With the PPOMS persistence mechanism there is no way of knowing whether an object once placed on a database is still reachable, since reference counts are not maintained. To introduce this would require significant changes to the system, and cause a decrease in performance that is unwarranted. Thus the metadata information can be a superset of the types currently reachable from the root of persistence. It is impractical to mark reachable objects so that metadata queries reflect the exact nature of the store, because of the inter-relationships between databases: to establish the persistent items in one database requires finding the transitive closure of all database primary sets. Besides being expensive, such an operation would not be possible while any other program had (exclusive) write access to a database. As the majority of metadata queries will involve determining type names and their properties, the fact that a superset of types may exist will generally be irrelevant.

6.2.5 CONTENTS

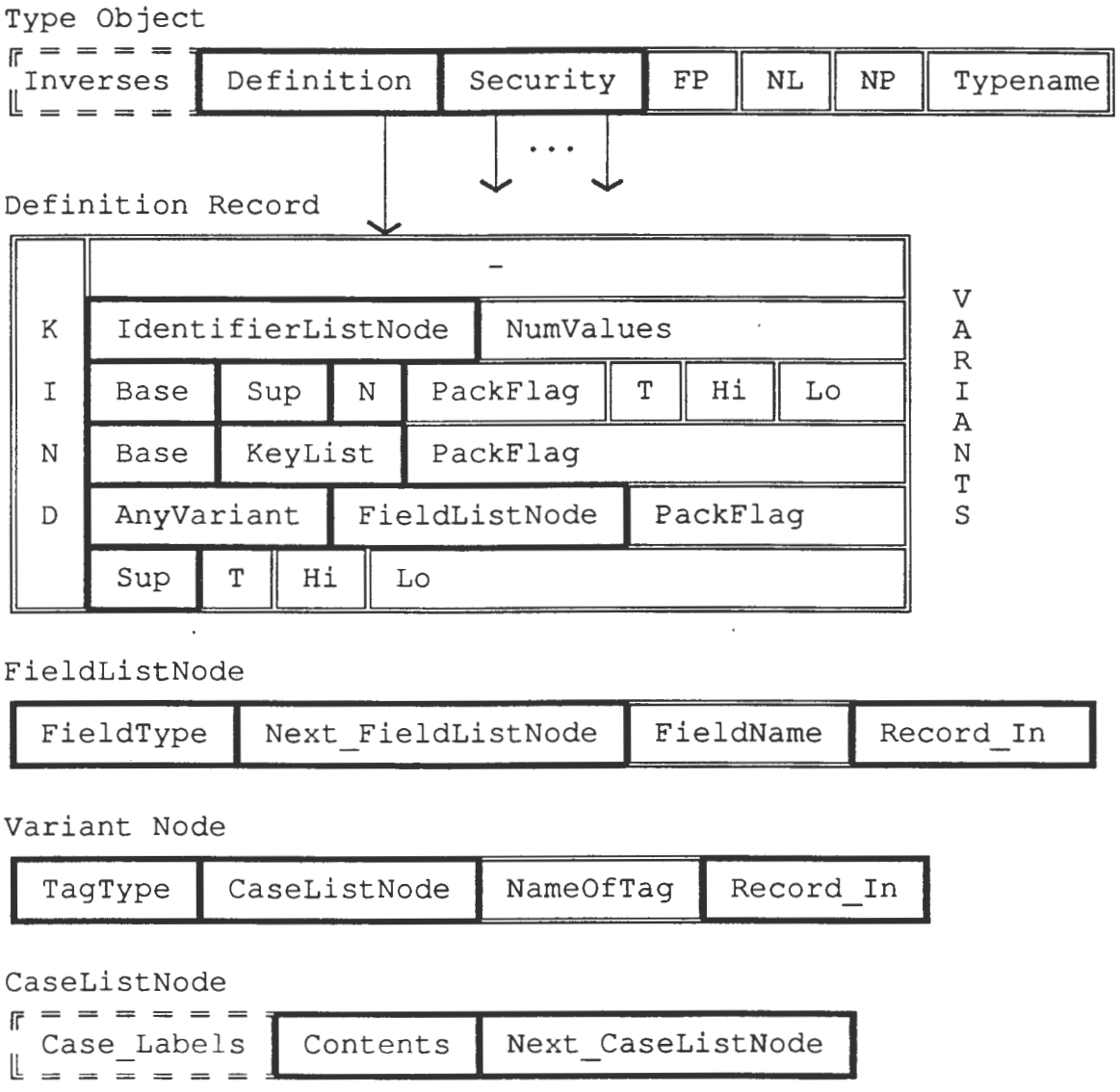
P-Pascal databases require some form of type object to provide for dynamic type-checking, type-clustering and data protection by type. Metadata currently incorporates type definitions, security, ownership, instance counts, cross references and placement information. Maintenance of cross references is optional; a CreateDb parameter indicates whether such information is to be maintained for a database. If there are a large number of inter-related types, the database designer may feel the additional space is unwarranted.

Standard types are included in the metadata so that all types are treated identically and queries are universally applicable. The presence of persistent objects of type Real (say) can be determined and its cross-references are obtainable in the same way as for user-defined types. Database objects must be of named type because they are reached through pointers, but they can have components of unnamed type. Thus some types in the persistent store can have only a definition and no name. These are inspected by queries relating to components, such as when finding all the places a specific type is referenced. For example if some field of a record is declared as ARRAY [1..10] OF TYP, this will be included in the cross-references to TYP. System types are included in the Metadata sets at database creation time.

6.2.6 USING THE METADATA

The system endeavours not to treat types as special objects. The availability of a bulk data type within the language provides a way of incorporating such information using conventional P-Pascal objects. Metadata is treated identically to conventional sets: it is a primary object "Meta" and can be accessed via an appropriate Locate. Standard set and record types give the definitions of the metadata sets and the records used to represent type objects, respectively. Programs can access metadata either using set operators to find subsets, perform joins, etc. or by calling standard metadata access functions.

All metadata is write-protected and some security-related information is read-protected. Primary sets "Ownership" and "Fields" provide information on type owners and record components respectively. Figures 6.1 and 6.2 illusrate the metadata record types. Standard functions for metadata querying allow the protection, ownership, population and existence of types to be determined. They also permit type component information and cross-reference data to be extracted. To avoid misinterpretation, metadata queries are specifically phrased with respect to a particular database, all open databases, the persistent store as a whole, or all currently reachable data (including transient types).
The functions are listed in Appendix C.

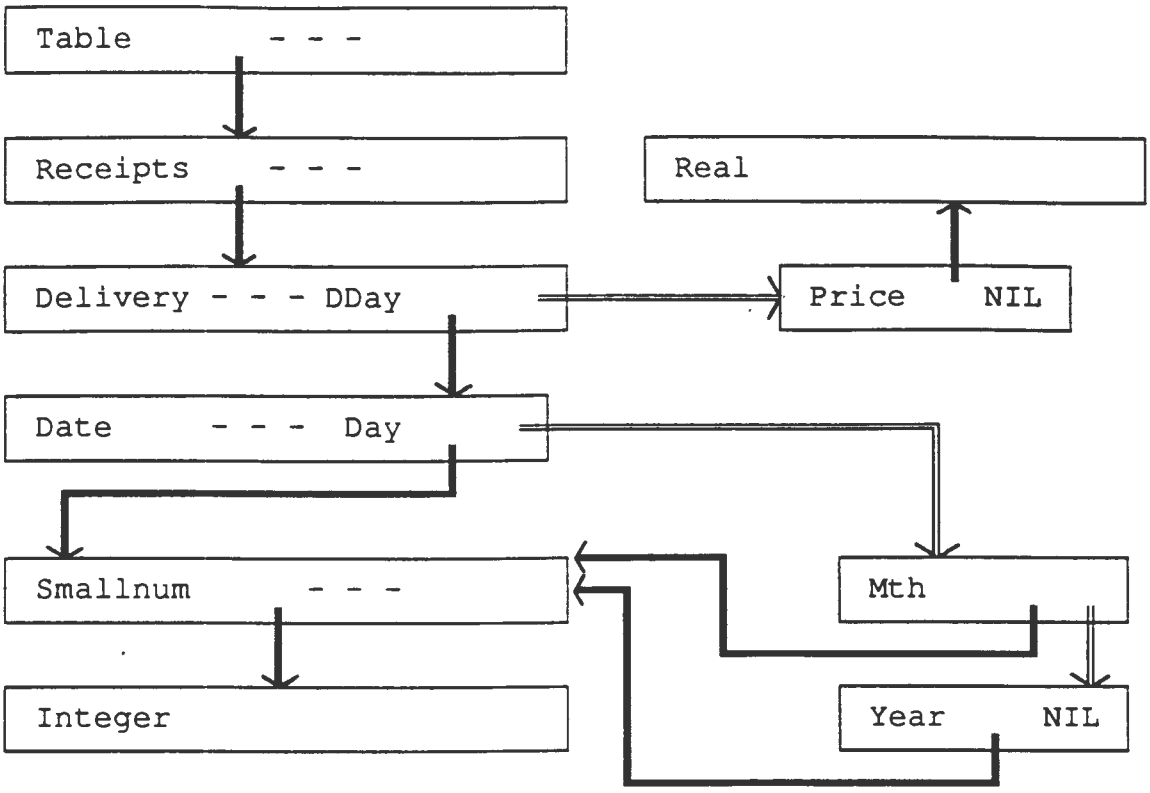


Definition cases, from top to bottom, are: Standard, Enumerated, Array, Set, Record and Subrange Types. Subranges are variants (simplified in diagram). The tagfield Kind is shown at the head of the record for clarity.

KEY: NP:NextPID, NL:NextLoc, FP:FirstPID, Base: pointer to base type; T: subrange kind (a tag field); Sup: type on which subrange is defined; N:pointer to Next subrange node, Record_In: inverse pointer to record of which this field is a part (not seen as a pointer by the PPOMS).

= = = = = = this field is a set
——————— this field contains pointer(s)

Figure 6.1 Metadata Storage (not drawn to scale).




```
TYPE Smallnum = 0..99;
   Date = RECORD
            Day,Mth,Year : Smallnum
          END;
   Delivery = RECORD
                DDay : Date;
                Price : Real
              END;
    Receipts = SET OF Delivery;
    Table = ARRAY [1..10] OF Receipts;
```

Figure 6.2 Graphical Structure of Type Definitions: an example.

6.3 REPRESENTATION

The choice of data structures for internal metadata storage is not straight forward, because the local-type to persistent-type mapping is many-to-many. This situation arises because type names can be reused in different program scopes, and type objects can be replicated on different databases. This section describes the data structures recording type information externally on the persistent store and internally within the interpreter.

6.3.1 REPRESENTATION ON THE PERSISTENT STORE

Type objects exist on the store, and pointers to these are included in object headers. This is necessary for dynamic type-checking, for clustering and security purposes, and to enable new types to be added to metadata during a Commit. These pointers are also used by the disk garbage collector to distinguish types in use from those to be discarded. The type information itself is stored as a network of metadata objects on the persistent store, with additional set structures for efficient random access to this data. The links between data and metadata are illustrated in figure 6.3.

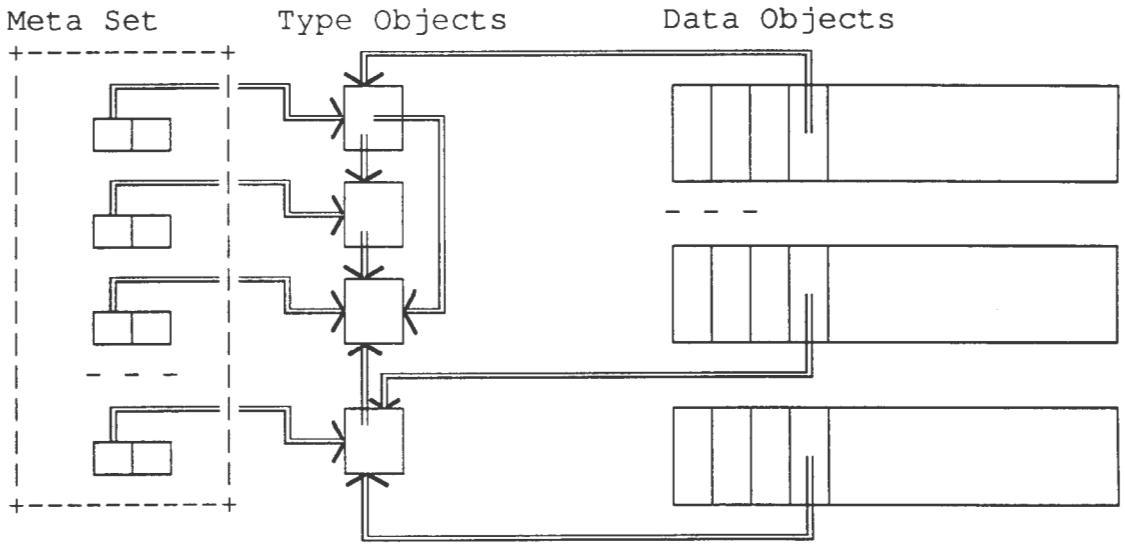


Figure 6.3 Type objects are referenced by data and metadata.

6.3.1.1 TYPE OBJECTS

A type object contains the type name, information on object placement, and pointers to its security data and its definition. Pointers are used to reference metadata which need not be retained in memory during execution, as well as optional metadata which can vary considerably in size. Type definitions are represented as a graph of P-Pascal records, as can be seen in figure 6.1. In addition to speeding up type-checking and saving space [Conn90], this ensures that a type T1 is not removed from a database during garbage collection if there is some type T2 defined in terms of it.

Cross-referencing information is more efficiently obtained when types are directly inter-related by pointers. To handle such queries, inter-type references are bi-directional. This is accomplished by attaching a set of inverse pointers to all type objects. These are not seen as pointers by the PPOMS, so they do not affect reachability. There is a separate copy of every type object on each database where it is used, to reduce access conflicts. Their definitions and security are identical, but each has different clustering data.

6.3.1.2 METADATA SETS

Type definitions contain references to component types, and object headers point to types to permit clustering and security enforcement. Type objects thus cannot be elements of a set, as pointers to individual elements are not permitted. Alternative strategies for keeping sets of type objects were considered. One possibility is for type references to take the form of a type name instead of a pointer. However, besides the decrease in speed, this complicates garbage collection of types. Alternatively, the set concept could be changed to allow pointers to elements from other objects. This is contrary to programming language principles that advocate transparency of implementation details (eg. encapsulation and abstract data types). A set is seen by the programmer as a single object; and it is inappropriate to permit pointers to parts of this, particularly since components of other structures cannot be referenced. The internal storage of sets would also need to be changed, making set processing less efficient. A third possibility is to use two kinds of type object; one comprising type name, definition and security (called type-definitions), the other containing name, owner and clustering data (called type-descriptors). The latter can be organised into sets, and can include a pointer to the corresponding type-definition. References to type-definitions are then possible, but to find the clustering/ownership data a separate access to the type-descriptor is necessary.

The approach adopted in P-Pascal does not have type objects as elements of any set, so other objects may point to these. Instead, "Meta" is a set with elements comprising a key and a pointer to the type object: i.e. <Typename, TypePtr> records. Such a set is termed an index-only set (see chapter 8). Each TypePtr points to a Type object which has all the type information. This in turn contains pointers to any other Type objects on which it is defined. Additional index-only primary sets named "Ownership" and "Fields" are also maintained on each database. Their elements are <Owner, Typename, TypePtr> and <Fieldname, Typename, TypePtr> respectively. The Ownership set permits efficient access to all type objects having a specific user as their owner. The elements of "Fields"

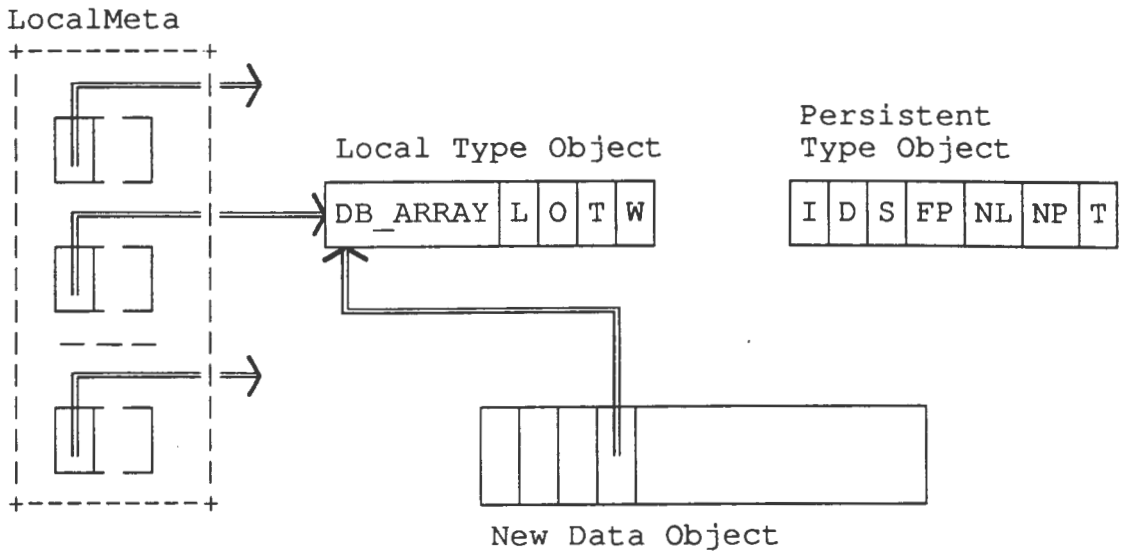
point to field-descriptor nodes, enabling eg. metadata on all "cost" fields to be easily obtained. For this reason, the fieldnode structures include an inverse pointer to the record type of which they form a part.

A system-owned database QQMETADB stores all type information for the persistent store as a whole, to maintain consistency between individual databases. This database has the added advantage of facilitating metadata queries relating to the entire store. When a new type is added to a database, QQMETADB is consulted to ensure that the definition of this type does not conflict with that in some other database. If the type does not exist in QQMETADB, it is inserted there.

6.3.2 LOCAL METADATA

The type objects and metadata sets used by the interpreter differ in structure and function from their persistent store counterparts. In addition, type declaration information is retained in the object code for checking against database definitions. These encode all the information in a type declaration: the type name, its definition and security data. On entering the program itself, as well as any procedure or function, any type declarations encountered are written to the object file. The encoding of types reflects the type representation employed on the persistent store. The only adjustments made are to replace pointers by the integer location of the corresponding type; and lists (and sets) by an integer N followed by their N entries.

Only the minimum metadata required by the runtime system is kept in memory. Every type declaration in the program is represented by a "local type" object, comprising the type name, the location of the type definition in the code (termed CodeLoc), flags indicating security authorisations (described in the next chapter) and Def_OK and Wrong_Def flags. The last two flags record if type-checking the declaration against its persistent store definition succeeded, failed or is pending. In addition a DB_ARRAY keeps the addresses of the type on each database (as database number and PID), to accelerate Commit. For rapid access to local types, the system keeps an index-only set LocalMeta referencing these objects. Its elements comprise <CodeLoc, TypePointer> records, providing an index on CodeLoc so that local types can be efficiently obtained for type-checking purposes. This organisation can be seen in figure 6.4.



T: Typename; L: Code Location; O: Def_OK; W: Wrong_Def; NL: NextLOC; FP: FirstPID; NP: NextPID; S: Security pointers; D:Definition pointer; I: Inverses.

Figure 6.4 Local Metadata Representation (not drawn to scale).

6.4 IMPLEMENTATION

Runtime metadata is used for type-checking program declarations against the store and for establishing security authorisations. This section outlines the type-checking algorithm and the effect of metadata maintenance on Commit and persistent store Garbage Collection. Some problems regarding database locking are also discussed. The effect of security enforcement on metadata handling is discussed in the next chapter.

6.4.1 HEAP OBJECT CREATION

The routine executing a "New" statement uses a compiler-supplied parameter giving the CodeLoc of the new object's type. The LocalMeta set is accessed, and the TypePointer extracted from this element is copied into the header of the new heap object.

6.4.2 TYPE CHECKING

The interpreter performs dynamic type-checking to ensure that program declarations match their counterparts on the store. To do so, it accesses the type descriptions encoded in the program and compares this with the graph representing the type definition on the

persistent store. A flag is set in the local type object to reflect the result. In addition, if the check is successful, the PID of the type object is recorded.

Dynamic type-checking in P-Pascal is required only to ensure that an object retrieved from the persistent store is of the type expected by the program accessing it, and that new objects placed on the persistent store associate the correct definition with a type name. We shall consider each of these in turn. The only way in which data can be retrieved from the store is by means of a Locate, or a Locate followed by pointer traversals. When a primary object is Located, its type must be checked against the program type declaration. Deep type-checking is performed: that is, if the primary is of type T and T is defined in terms of T_1 ... T_N, then each of these N types are checked recursively. Once a primary type has been verified in this way, there is no need for further type-checking when following pointer chains from this object. A program can access a structure even if part of its definition is incorrect; the security subsystem is employed to hide the affected data and prevent access to it.

Lazy type-checking is not used, since programs could then retrieve an object from a database even if its components were incorrectly defined. It is in any event impractical to delay the type-checking of embedded values; only the type-checking of pointer components could possibly be delayed until the pointer is dereferenced. However, lazy type-checking would require that every PID dereference carry the overhead of testing whether type-checking is required. In addition the interpreter would have to be informed of the type expected, since it cannot be determined statically which pointer traversals will access persistent data (except possibly by means of program analysis, but at considerable cost). Furthermore, the mapping from PID to typename is many-to-many, so a considerable amount of additional type information must be maintained by the interpreter and altered when scope changes.

The ability to type-check only on Locate, and not on pointer traversals, means the points at which type-checking is required are known statically. This enables the compiler to supply the code-location of the primary type as a (hidden) Locate parameter, for efficient access to the local type object via the LocalMeta set. The type-checking flags determine if any further action is required. If Wrong_Def is set, a type-mismatch error is returned immediately. If Def_OK is set, then name equivalence and uniqueness of type names on the persistent store together guarantee that no further checking is necessary. The type

PID in the primary object is entered with the database number in the DB-ARRAY (unless this has been done previously).

On the first access to a persistent type, the type object is fetched from the persistent store. If its type name differs from that of the local type, type checking fails. Otherwise, deep type-checking against the program type in the object code is performed, stopping whenever a type is encountered that has already been flagged Def_OK or Wrong_Def. Any difference in type definitions causes an error to be returned, and the Wrong_Def flags to be set. Otherwise the graphs (and security information) are disposed of, Def_OK is set and an entry is made in the DB_ARRAY.

A new object X can only be written to the store if some persistent object points to X, if X is Saved, or if X is the descendant of a new primary. If X is referenced by an existing persistent object, then its type will have been checked before its parent was accessed. Therefore, the only additional type-checking required is deep type-checking of new primary objects. This can be done on either Commit or Save. The former is more efficient, as Commit involves finding the type of such an object in any event, to obtain its TYPE_PID.

6.4.3 COMMIT

For newly persistent data, Commit follows the type-pointer from the header to the local type object, to perform type-checking and obtain the corresponding type PID. Its action depends on the type-checking flags. If Wrong_Def is set, the Commit aborts. If Def_OK is set the PID of the type object on the target database is taken from the DB_ARRAY. If the type has not yet been found on that database, it is now located in its Meta set (accessing this on TypeName). The associated PID and database number are added to the DB_ARRAY for future use. If the type is not found in this Meta set, it is copied from another database (using any address in the DB_ARRAY) onto the target database and included in its metadata. In this way types are consistent across the store. As the metadata insertion occurs during the first step of a Commit, this includes all new type information when it is written back in the final step.

If the program type has yet to be compared with its persistent store counterpart, this is done in the same way as for Locate. Any difference in definition causes the types to be flagged and the Commit to abort. If a type is totally new to the persistent store, it is

inserted in both the target database's Meta set and the QQMETADB database. The use of QQMETADB raises questions concerning sharing and mutual exclusion, which are addressed below.

6.4.4 DATABASE LOCKS

The limited form of concurrency in P-Pascal has implications for global dictionary maintenance. Either one writer or many readers may access a database concurrently. Thus programs should minimise the number of databases they access; and in particular should limit the databases to be updated. For this reason, the types in use on a given database are described by type objects on that same database: this enables all the necessary typing information to be extracted independently of the locks on other databases.

QQMETADB serves as a central repository for all persistent types. It ensures that types of the same name are defined (and protected) identically on different databases. To add a persistent type to a new database it is generally copied from another open database. QQMETADB has to be accessed instead only if programs create primary objects and do not use existing data of that type. Several databases can have existing persistent types added to their metadata concurrently, requiring at worst only read access to QQMETADB. When an application writes to QQMETADB (i.e. introduces a new type to the persistent store), no other run-unit can access this database. However, there is no need for a program to have access to QQMETADB unless it intends to copy a type to a new database. Thus a program can use the persistent store even while new types are being incorporated, if the program does not add types to any database. An investigation into concurrency is beyond the scope of this work, and has been conducted eg. by Wai [Wai].

Alternative interpretations of metadata scope were considered to see if a viable scheme was possible which avoided the contention problem. The alternatives are:

(A) Type names can be re-used at will in any context. This is not possible under name equivalence. It would in any event require a great deal of additional dynamic type-checking, since every object retrieved from the persistent store would have to be checked to ensure that its definition of a type agreed with that expected by the program.

(B) Types are unique on a given database. This causes problems when objects on this database reference data in another database with different definitions for types.

(C) Types are unique across inter-linked databases. This can cause merging problems when two databases are linked for the first time and their metadata conflicts. This strategy was abandoned because of the complexity it presents to the user of the persistent store, especially in view of the fact that the database on which an object will be placed cannot always be determined.
(D) Types are unique across the persistent store. This is the way the system was implemented. Its only disadvantage is potential conflicts arising if concurrently executing programs attempt to add a type to the persistent store, as discussed above.

6.4.5 GARBAGE COLLECTION

When a type T is no longer reachable from a database root, it is discarded by the Garbage Collector in the same way as any other unreachable object. The pointers from metadata elements to type objects do not cause type objects to be marked. Furthermore, the metadata sets are updated to reflect only types which are still persistent. This requires altering the reachability phase of Garbage Collection.

The metadata primary objects are the last to be considered in the reachability phase. In an initial scan of the "Meta" set, the type to which each element points is located; the element is deleted from the set if this type object is unmarked (unreachable). Thereafter, reachability is applied to the "Meta" set in the conventional way. Sets "Ownership" and "Fields" are handled similarly.

6.5 CONCLUSION

6.5.1 SUMMARY

An early P-Pascal design decision was to avoid a schema constraining persistent data storage, and maintain metadata automatically instead. This chapter described the type information available and the ways in which it is stored, accessed and updated. The type-checking algorithm was outlined, as well as the effect of metadata management on Commit and Disk Garbage Collection.

6.5.2 DISCUSSION

The organisation of the persistent store as a number of interlinked databases complicated metadata management because references are not limited by the unit of locking (the database). Thus it was not meaningful to make this unit define a scope for type names; types had to be unique across the entire store. Any system which modularises the persistent store in some way and allows references between modules will incur the same problem.

Some P-Pascal features were particularly suited to metadata processing. The uniqueness constraint on type names in the store is consistent with the name equivalence model of type-checking. Pointers are typed, so eager type-checking is possible: entire primary structures can be type-checked at once, and dynamic checking is limited to Locate and Commit operations. A bulk data type is useful for non-procedural access to metadata, and facilitated the handling of inverses, variants, local metadata and persistent types. In particular, the index-only set is invaluable for random-access to type objects which remain independent entities, capable of being referenced by other objects. On the other hand, the P-Pascal clustering and type-related security facilities make additional demands on the metadata subsystem. Without these, type information would only need to be attached to the primary directory. A discussion of the effect of name equivalence on typical metadata operations can be found in [Conn90]. The range of metadata and the need to query this information increases in languages with richer type systems, eg. Galileo and Napier. The structures and techniques employed above can be extended to handle constraints, type hierarchies, first class functions and polymorphism.

CHAPTER 7 SECURITY

The type system of a programming language is one means of protecting data from misuse. The privacy of confidential information and the prevention of unauthorised data alteration are equally important in most database applications. This chapter discusses the implementation of security in a persistent programming environment by considering the following questions:

How does a persistent environment differ from a traditional database environment?
What security should be provided?
How is security information specified in programs?
When does security enforcement take place?
How should the persistence system implement this?

7.1 DIFFERENCES FROM CONVENTIONAL DATABASE SYSTEMS

In a conventional database environment, database data is distinct from program objects. In a persistent environment types and values can be either transient or persistent, and there are situations where security violations cannot be detected until Commit. The traditional database operations find, delete, update and insert do not exist in a persistent programming language, and operations on transient data cannot be distinguished from database manipulations. This has implications for the enforcement of corresponding access rights. Individual objects called "primary objects" can be accessed by name (via the Locate statement) and there can be several paths to any one database object. This raises the question of how such objects should be protected.

7.2 WHAT SECURITY SHOULD BE PROVIDED?

7.2.1 POSSIBLE SECURITY FACILITIES

Security involves the protection of database items from illicit disclosure or manipulation. A security policy is characterised by its granularity, access rights, authorisation mechanism, violation actions and its method of security specification and alteration. Some alternatives are outlined below and then the P-Pascal system is described. Detailed discussions of database security can be found in [Denn79, Fern, Hsia].

7.2.1.1 GRANULARITY

The granularity of the object for which a privilege can be granted can vary from entire databases to a specific value in a particular object (eg. John Doe's salary). Data can be protected by type, and authorisation can depend on an access predicate. The latter can enforce the hiding of amounts exceeding a certain value, restriction to the user's own personal records, etc. Statistical control permits the application of operators such as Average to a collection of data while denying access to individual values [Beck80, Chin, Denn78, Denn80, Yu77]. Protection of special programs or storage devices, and data encryption [Lemp], are also methods of security enforcement.

7.2.1.2 RIGHTS

The kind of access permitted by an authorisation can range from general clearance to specific privileges such as the ability to perform only certain operations, at certain times, on certain days, from certain terminals. Security systems prevent unauthorized data reading ("information theft"), modification and destruction. Information destruction has many forms; deletion of objects, types, indexes and access rights are distinct privileges. Similarly, several modification rights can be distinguished: the insertion and update of data, types, indexes and security information.

7.2.1.3 AUTHORISATION

Three kinds of security clearance mechanisms predominate in database management systems. The first involves each user having a unique identity, with authorisation being granted or withheld according to the individual issuing the request [Conw]. Secondly, objects can be protected by secret passwords or procedures. The third method, data classification [Wood], involves the ranking of users and data so that each is placed at some level in the security hierarchy. For example, there could be levels "Confidential", "Secret", and "Top Secret"; or there could be a large number of levels, reflecting the many positions in the user hierarchy.

7.2.1.4 SECURITY SPECIFICATION AND ALTERATION

Security can be established in the declaration section of a program, by means of executable statements or using a separate utility program. The use of a separate utility

with exclusive access to a database prevents protection changing in the course of an execution, with the result that clearance need not be repeatedly determined.

7.2.1.5 SECURITY VIOLATIONS

A violation occurs when an operation is attempted without authorisation. Actions can range from logging illegal attempts to terminating a program. A single policy can apply to all such situations or the action required when a violation occurs can be specified with each privilege.

7.2.2 P-PASCAL SECURITY

The P-Pascal security system has been kept as simple as possible, in order to focus on the practicality of such a facility in a persistent environment. It is presented by considering each of the aspects discussed above.

7.2.2.1 GRANULARITY

In P-Pascal, security can be associated with whole databases, individual objects, entire types, fields of individual objects or fields of record types. As an example, a particular Student record in a database could:

- be protected in its entirety, because the entire database is protected.
- be protected in its entirety, because that Student (J Jones) is protected.
- be protected in its entirety, because all Students are protected.
- be protected in its entirety, because all paths to this Student object are protected.
- have only its Mark field protected, because that Student (J Jones) has a secret Mark.
- have only its Mark field protected, because all Student Marks are protected.

As protection of specific entities is only possible for uniquely identifiable objects, only named objects in P-Pascal can have this kind of security. This collection of objects is the set of primary objects; security can be specified in a Save, Drop and Locate. This level of protection is made possible because the persistence mechanism is based on reachability, rather than persistent extents. The authorisation information is treated identically to other objects, with the same security mechanisms to protect it.

Some security features which have been designed but not implemented, are mentioned below. The presence of pointers in a database does not lend itself to security specification in terms of an access predicate: it is not easy to ensure that an unprotected object is always reachable, particularly when there is more than one predicate applicable to this object. Sets are relatively easy to protect in this way. Statistical control is also easily applicable to sets. Standard set functions (to find the average, minimum, total, count, etc. of a set of values) can include a password parameter, and be applied to any set irrespective of the read-authorisation on the individual values. User-defined protection of type objects has not yet been implemented; security alteration privileges are currently limited to the owner (creator) of a type, and a single metadata disclosure policy applies to all types.

A type T has the same definition on all databases where it is used, so that a program can freely store such objects on any open database. For the same reason type protection is applied to the entire persistent store. Were this not so, a run-unit without security clearance would succeed in making such objects persist only if they happened to be stored on the unprotected databases, and would fail otherwise. A program with the appropriate security clearance could write objects to several databases without being aware that the data was unprotected on some of them.

7.2.2.2 RIGHTS

In P-Pascal two levels of data security have been implemented: read- and write-protection. The former prevents unauthorised disclosure of information, the latter permits access but precludes unauthorised change, creation and deletion. Lack of read-access to an object prevents any reference to that object irrespective of write-authorisation.

Operation-related security specification is natural within a database environment, but not in a persistent programming language, where the equivalent of "insert", "find", "update" and "delete" do not exist as separate statements. There is currently no privilege allowing an individual to create or destroy authorisations: only the owner of a type or object can specify the security on that data. Permission to create a new primary object or a new type could be associated with a database; however it is felt that this is contrary to the spirit of a persistent programming language which is to be flexible in terms of additions to the store.

7.2.2.3 AUTHORISATION

A hierarchical security structure is not sufficiently flexible, and is suitable only for application environments where the data structures are fairly rigid and static, eg. in the military [Date]. In the initial design, privileges were specified as sets of user authorisations. This was implemented by attaching a set of privileges to each type and primary object. A privilege object comprised a user-id and a set of authorisations applicable to that user: each indicating field name or entire-object, and read or write mode. This scheme was tailored to the runtime system requirements, so that all the rights of an individual to the primary or type could be established simultaneously.

This was subsequently replaced by the password or "lock and key" approach. When authorisation is formulated by listing privileged users, the introduction of a new type has to be done explicitly with the aid of a separate utility. Otherwise the new type is either totally unprotected or totally inaccessible to all other users. This scheme also requires a considerable amount of storage space. P-Pascal aims at limiting the use of utilities, so privileges can easily and naturally be specified within programs. Maintaining lists of authorised users makes security enforcement enduser-oriented rather than program-oriented. In P-Pascal, authorisation is a property of a program or procedure, not of individual users; it is determined by the task being performed, rather than according to the individual executing this task. To restrict the users who may run a program, should this be necessary, is a separate issue to be handled by the program itself. The details of password protection are presented below.

7.2.2.3.1 READ PROTECTION

Without the readkey of a type, a program cannot reference such data in any way. If the readkey associated with any component of a type is missing, the object may be manipulated as a whole and the values of its other components may be accessed in the normal way. Any reference to the protected component causes an error.

In contrast, if the readkey of a primary object component is missing, neither this component value nor the object as a whole may be referenced at all; only the values of other components are accessible. This prevents a program copying the entire primary to another object of the same type, to examine the protected field. Name equivalence

ensures that this extra precaution need not be taken in the case of type security: if an entire object is referenced, it can only be assigned to another object of identical type; hence the protected value will not be accessible there.

To access an object, a program requires the readkey associated with both its type and its "access path". Access path protection means that the correct readkey must be supplied when a primary object is Located, while an object reached via a pointer dereference requires that the program supply the readkey for that pointer. If the pointer is a field of some record, with its own component password, this must also be correct. As a result, if a program correctly specifies the keys of a type, it can access such objects via any authorised path, irrespective of its rights to access the object via other paths. In particular, a protected primary object which is also reachable from another database structure may be processed via this alternate route (with appropriate authorisation), while direct access to that object in a Locate is denied if its readkey is not specified. For example: if the Jobs which an Employee controls is represented as a read-protected field of the Employee record, they cannot be inspected without supplying the associated read password. If these same Job objects are also part of a separate list of Jobs however, then they can be accessed via this list if the program is authorised to do so.

If a primary object is accessed via some pointer, the primary is subsequently treated as if only its readkey was supplied; that is, if the object is write-protected, or if any component has any associated security keys, these privileges are denied (since the corresponding passwords have not been supplied). This can be circumvented by a program with appropriate authorisation. All that is required is to Locate the primary object and supply its passwords; either before or after the object is reached via the alternate route.

7.2.2.3.2 WRITE PROTECTION

Failure to provide the writekey of a type or primary prevents any changes to any (part of) such objects. If the writekey of a specific component is missing, other components may be assigned new values - but this component itself, as well as the object in its entirety, cannot be changed. In particular, write-protection of a pointer component implies that the referenced object cannot be destroyed - that is, the pointer cannot be altered to NIL to make the target object unreachable. In the case of a read-only pointer component, the protection does not prevent changes to the referenced object itself. It signifies that a new

object cannot be referenced from here; but the value(s) in the referenced object can be altered.

The alternative of descendants inheriting write-protection was investigated. That is, if a pointer was read-only, the system would flag the object it referenced as read-only as well. This raised the question whether such data should be writable if reached via another path through the persistent store, to which the program has write authorisation.

Consider objects O_1, O_2, ..., O_m each containing a pointer to the same object X. X is of type T and the program is authorised to write to T objects. Suppose that the pointers in O_1 .. O_k are write-protected and those in O_{k+1} .. O_m are not. Possible protection strategies are:

(1) Since X is on some write-protected path (in fact it is on *k* of these), it is write-protected.

(2) Since X in on some path to which the program has write-permission (in fact it is on *m-k* of these), it is not write-protected.

(3) The write-authorisation on X is determined by the last path through which it was reached. Thus, if the object was last accessed from O_1 ... O_k, X is write-protected; otherwise it is not.

(4) Write-protection is not inherited from any parent, so X may be written.

Strategy (1) cannot be enforced unless a counter is included with every object, indicating the number of write-protected paths to that object. This is too expensive as it requires checking counters every time such a pointer is altered: the current value must first be examined, and the count of the target object decremented.

Unfortunately inherited protection as in (2) above is easily circumvented and as such is totally ineffective. If type T is defined having a write-protected pointer of type T2, its T2 descendants can easily be written to by any program, irrespective of its authorisations on that pointer. The program simply creates a new primary object Y of some type T3, where T3 includes an unprotected T2 pointer component (T3 can be a new type if necessary). Y is then made to point to a T2 descendant of a T object. The target objects are now writable by virtue of the newly created path, as shown in figure 7.1. The new objects can subsequently be destroyed in order to prevent detection. A similar argument can be used to show that (3) is equally ineffective. For this reason, method (4) has been used.

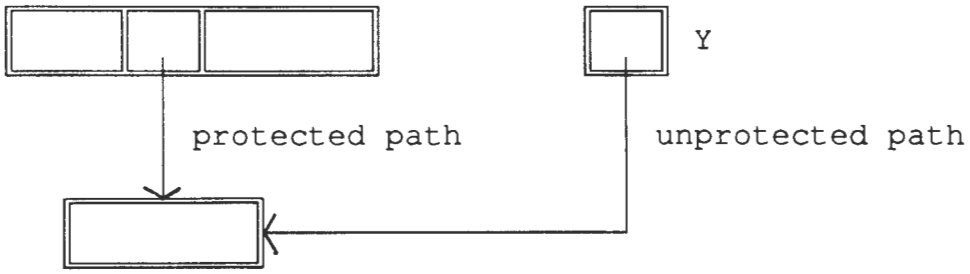


Figure 7.1 Write-protection of pointers cannot prevent target object alteration.

In practical terms, this means that the contents of a referenced object can be altered, but not the object itself. For example, suppose Winner is a protected field of some object, and points to a Person instance. The pointer cannot be changed without the necessary write-authorisation to Winner; but the contents of that Person (eg. Name, Address, etc.) can be updated. If Person objects are also part of an ordered data structure, the Winner cannot be changed from one person to another (swopping contents will cause the data structure to become unordered). Otherwise the change can be made; but it will cause all relationships of the affected object to refer to the new individual, and hence is unlikely to go undetected.

In a relational system, such a Winner attribute would contain the key of some Person tuple stored elsewhere in the database. Changing the Winner values would involve associating this identifier with a different person in the Person relation. If user-visible keys serve as identifiers (rather than system-controlled surrogates), the change is not possible, since eg. a person's Social Security Number cannot be changed. Otherwise, the situation is analogous to that of P-Pascal; the change would automatically cascade through to all relationships and thus detection is highly likely.

7.2.2.4 SECURITY SPECIFICATION AND ALTERATION

In P-Pascal, type passwords are given with their declaration and primary passwords are supplied as parameters of Locate and Save. Security can be specified for any type, irrespective of whether or not persistent instances are manipulated by the program. Primary security is established when an object is Saved on a database. Each Locate resets the program's rights to the object. The keys associated with a type are fixed at the time the type is first introduced to the store, and can only be altered subsequently by running a utility program with exclusive access to the database. The only user that can

change the security associated with a type is its owner who first caused such a type or primary object to be placed on the database.

7.2.2.5 SECURITY VIOLATIONS

When a read-violation occurs, an error is raised; a write-violation aborts Commit. Logging such attempts has not yet been implemented. The action to take should ideally be a parameter of database creation, to suit the needs of the application environment.

7.3 HOW IS SECURITY SPECIFIED IN PROGRAMS ?

7.3.1 PRIMARY PROTECTION

The functions defined for primary objects include security specification parameters. These are:

Save(database,name,primaryptr,readkey,writekey,componentkeys)

Locate(database,name,primaryptr,readkey,writekey,componentkeys)

and Drop(database,name,readkey,writekey,componentkeys)

where the last parameter, or the last two or three parameters, can be omitted. These specify the read- and write-password of the object, and those of its components, respectively. Any string literal or variable can be given as a password. The use of variables permits security clearance to be tailored to the program's end-user if required. Component keys are specified as a set, since this provides a convenient way of passing an indeterminate number of parameters. Each element is a string comprising mode, fieldname and associated password, concatenated in this order and separated by colons. An example is given below.

```
{ Program 1 creates a primary object with confidential parts }
{ field "field1" is read-protected, password "rdkey1"; etc.}
fldsec = [ "R:field1:rdkey1", "W:Field2:wkey2", ... ];
Save(db,"strucname",ptr,"rdkey","writekey", fldsec);
{ A protected primary object has now been stored on the database.}
```

```
{ Program 2 uses primary object, supplying some passwords }
fldsec = [ "W:Field2:wkey2", ... ];
Locate(db,"strucname",ptr,"rdkey","writekey", fldsec);
```

7.3.2 TYPE PROTECTION

A type declaration can optionally be followed by READKEY <string>, WRITEKEY <string>, or both. These reserved words can also be used to introduce protection for individual fields of a record.

Example:

```
TYPE T = ARRAY [1..N] OF INTEGER READKEY "fred";
        T1 = RECORD A:REAL;
                        B,C:REAL READKEY "tom" WRITEKEY "dick";
                        D : T WRITEKEY "harry"
                END READKEY "T1secret";
```

etc.

The writekey on D applies only to this field of T1 occurrences, and is termed component security. The readkey on T applies to all T instances, not only those occurring as the D field of a T1, and is called type security.

7.4 RAISING SECURITY VIOLATIONS

This section identifies the points at which security enforcement takes place, and explains why. Failure to supply the correct readkey for a type must raise an error as soon as a PID pointing to such an object is dereferenced. Similarly, a Locate lacking the correct readkey must raise an error. In addition, any reference to a read-protected field must be detected immediately.

A new object cannot be Committed to the persistent store if the writekey of its type, or any component thereof, is missing. If a new object is created and its type is present on some database, write-authorisation cannot be enforced immediately, as this object may never be made to persist. To trap a security violation as soon as a new object is created,

types would have to be declared transient or persistent; which is contrary to the goal of persistence independence [Atki83]. The P-Pascal system only reports a write-protection violation (and aborts Commit) if and when the program attempts to make such an object persistent. This is analogous to the detection of a write to a read-only database during Commit.

An illegal attempt to write to an object retrieved from the persistent store can either be trapped directly or on Commit. The former approach would permit error messages to identify the offending program statement, and a violation-flag could be set for checking at the start of a Commit. However, security enforcement on every read, assignment and VAR parameter passing is costly, particularly in the case of protected fields. Write violations are therefore only checked on Commit. This is consistent with the treatment of new objects created without authorisation, and the type-checking of new primary structures. The error record returned by Commit identifies the primary or type (and field name, if any) for which the writekey is incorrect.

This section has discussed the dynamic enforcement of security constraints. Authorisations could be established statically by accessing the persistent store metadata, and generating appropriate instructions for all references to data for which some privilege is missing, but this is not practical for two reasons. Firstly, it cannot be determined statically whether a data object is persistent or not. Hence code would have to be generated on every access to data without appropriate type authorisation, so that a violation could be raised in the event of the object being persistent. This is far more costly than simply checking an object when it is fetched from a database or Committed to the store. In addition, to protect primary objects as described above, data flow analysis of the program would be required to distinguish references to primaries.

7.5 IMPLEMENTING SECURITY

Before outlining security implementation, the external and internal data structures used by this subsystem are described.

7.5.1 SECURITY REPRESENTATION ON THE PERSISTENT STORE

Two types of security are enforced in P-Pascal: primary protection and type protection. The elements of the Primary Set include pointers to security information. Each element

comprises the primary object name and six pointers: to the object, its owner, readkey, writekey, component-readkeys and component-writekeys respectively. This is preferable to adding security pointers onto the actual objects, making them differ from other objects of the same type. For a type, the first pointer references its owner and the next four point to its password information: type readkey, type writekey, component readkeys and component writekeys. Any security changes are automatically applied to all instances of the type. The adding of a new type to the persistent store during Commit causes its security information to be recorded along with it.

Password information is represented identically for primaries and types. The read and write passwords and the ownership are three separate strings. The component readkeys are stored in a list of records comprising offset and password. The list is in ascending order of offset. Component write passwords are represented in the same way. This representation is more compact than a fixed size structure with two pointers per field, pointing to readkey and writekey respectively (or NIL). The fixed structure occupies less space only in the relatively rare cases where the number of passwords exceeds the number of fields. Concatenation into a single demarcated string is also unsuitable, since a type is a P-Pascal object and must be of fixed size. Sets of offset-string pairs for readkeys and writekeys require too much space, which is unwarranted as set operators are not used by the security routines.

7.5.2 RUNTIME STORAGE OF AUTHORISATIONS

The type descriptions in the object code include the security strings associated with a type, in addition to type name and canonical form definition. Security clearance is established when type definitions are checked. Passwords are not retained in memory, but flags are kept to record authorisations.

7.5.2.1 DATA OBJECTS

Four bits of the object header are used for security enforcement: the read-only, part-protected, part-hidden and is-primary flags. These flags are set whenever a persistent object is fetched from disk. In the case of a primary object, the primary security as well as that of its type are examined in order to set the flags; a particular bit is set if such access is denied by either. The flags are updated each time a Locate occurs.

A security flag is set if the corresponding privilege is withheld; in this way flags of non-persistent data such as variables can simply be zeroed. The read-only bit is set if the entire object is write-protected. Part-protected is set if the program has write access to the type, but the writekey of a component is missing. Hidden-fields is set if read authorisation for some component is denied. Is-primary indicates that the object is a primary and thus its type- and primary-authorisation have been combined.

7.5.2.2 RECORD TYPES

The last 2N bits of a local type comprising fields store the security clearance associated with its components. The first N bits indicate read-protection, the next N give write-protection. In each of these sections, there is one bit for each word in the record; the bit is set if the corresponding privilege is denied. Where a field occupies many words, all the bits are set - not only the first - to cover the case where a field is itself a record. These flags are similar to the Constancy Bitmap employed in the Napier88 Abstract Machine [PAM]. However, they are only meaningful for memory-resident types. The information is thus kept for local types only. Since local types are distinct from persistent store types, this data is not part of type objects on the persistent store.

Readflags precede writeflags to facilitate the mapping of an offset onto a corresponding bit. If the offset of a field is F, its read-flag is in word W =(F + 32) DIV 32, where offsets are numbered from zero (the most significant word of the record). The bit to use in word W is given by B = F modulo 32. If the offset of the given field is F, its write-flag is in word W = ReadFlagSize + (F + 32) DIV 32, where ReadFlagSize is (D + 31) DIV 32; D being the size of the record (excluding header). The bit to use in word W is again given by B = F modulo 32.

7.5.2.3 RECORD OBJECTS

To enforce component read-protection, three possible strategies were considered:

(1) replace the field value by zeroes when the object is fetched, restoring its original contents before writing the object back to the database;

(2) attach read-flags to all record occurrences so they may be checked when the value is accessed by the program;

(3) attach read-flags to record types, and check accesses by obtaining the type object.

The first strategy is most efficient, but unacceptable because it can give the false impression that the value is indeed zero. The last suggestion is time-consuming, since accesses to partly-hidden records require twice as many pointer dereferences. Although overall performance is primarily a function of disk accesses, it is nevertheless desirable to avoid this overhead. The second strategy is therefore preferable, being a reasonable comprise in terms of space and time. The flags require a negligible amount of extra memory in return for a reasonable saving of time.

Every record and every array of records has read-authorisation flags appended to its data. These flags corresponded to those of their type objects: there is one bit per word of data. The flags are part of the structure, and are retained when the object is moved between the local and the persistent heap. Although they have no meaning on a database, their retention simplifies fetching and committing, by keeping the object size fixed. Write-protection of fields is not carried with objects in this way, in order to save space. The policy of deferring write-authorisation checks until Commit makes this possible. Thus the security flags increase the data size by only 1/32.

The word in which the required read-flag is stored, is given by

W = S + (N DIV 32)

where N is the word offset of the field and S is the start location of the read-flags, which is calculated from the total size in the header (say T) :

S = (32*T + 1) DIV 33

The bit within word W is given by

B = N modulo 32.

All objects comprising fields have read-flags appended, even those for which no passwords have been defined. This ensures that primary objects have the same format as other objects of their type, since primary component protection may exist when type component security is lacking. It also permits the introduction of field passwords for a type when instances already exist on a database.

7.5.2 SECURITY ENFORCEMENT

The security bits of a type object are set when the type is first fetched from the database for type-checking purposes. Those of a primary are set when an object retrieved from the persistent store has its is-primary bit set. If the definition of a type T references a type T2, T2 will be type-checked at the time that T is checked ("deep" type-checking). At the

same time, security clearance for the referenced type T2 is established. Thus if any T2 value is protected from use by the program, the corresponding T components will be flagged accordingly. In this way protection of embedded types is maintained without requiring that nested records be represented by separate objects with their own header.

When an object is fetched from the persistent store, its security flags - read-only, part-hidden, part-protected and component read authorisation (if any) - are set according to the local metadata on its type.

7.5.2.1 CHANGING SCOPE

Before the introduction of security, a single LocalMeta set was adequate. To accommodate password alteration on changing scope, either this set can be updated on every routine call and exit, or it can be replaced by a list of sets (similar to the way scoping is done with a static chain). In order to investigate the utility of sets, the former approach has been used. Adjustment of LocalMeta requires deleting types no longer in scope and inserting new entries. The affected types can be determined statically or dynamically. If done statically, this complicates separate compilation. The modifications are thus done dynamically, and incur negligible extra cost compared to static determination.

The LocalMeta set changes from being an index on CodeLoc to an index on type name. This is possible because type names are unique in any scope and LocalMeta is now altered to reflect current scope. The change in LocalMeta is necessary so that names being reused in a nested procedure can readily be identified. The only side-effect of this alteration is that the compiler-supplied parameters to New and Locate identify types by their names rather than their locations. An aditional index-only set, LocalSecurity, is maintained for efficient access to local types given a type PID. This enables the protection of persistent objects to be readily determined.

The base types of these two sets are identical, comprising <TypeName, TypePID, Status, TypePointer> records. More than one element can thus have the same TypeName and TypePointer; however this redundancy is worthwhile because of the advantages of having a LocalSecurity set which can easily be updated on scope change. The four Status bits indicate if instances of the type are readable, writable, partly-readably and partly-writable respectively. An AllMeta set contains an element for every type declared in

every scope. This comprises <CodeLoc, TypeName, TypePID, Status, TypePointer> records and allows type objects to be readily located on procedure/function exit; so that they can easily be inserted in LocalMeta and LocalSecurity.

To facilitate the updating of metadata sets when scope changes, the activation record of each routine includes two sets of <TypeName, TypePID, Status, TypePointer> elements, named New-types and Old-types. On entry, each of these sets is empty. The code begins with a TypeDecl instruction for each local type, giving its name and CodeLoc. The interpreter moves any element of the given name from the LocalMeta set to the Old-types set and adjusts LocalSecurity accordingly, before inserting the new element in LocalMeta, LocalSecurity and New-types. Information for the new local type is obtained from the AllMeta set (accessed on CodeLoc). If it does not exist there, this is the first call to this routine. A local type object is constructed and an element referencing this is inserted in the LocalMeta, LocalSecurity, New-Types and AllMeta sets. On routine exit, the LocalMeta and LocalSecurity sets have all New-types removed and all Old-types restored.

7.5.2.2 ESTABLISHING OBJECT SECURITY

Whenever a persistent object is accessed, the security privileges associated with its type must be ascertained so that the object may be protected accordingly. The protection of an object is determined by the scope in which it is retrieved from the store, and it can only be used accordingly in other scopes. To implement any other strategy requires resetting security on every pointer dereference, which is clearly too expensive. An object can be Committed in any scope with appropriate type authorisation. The protection in force at the time it was created could determine write authorisation instead, but this means checking permission on every write to a heap object rather than only on Commit, which is significantly more costly.

Establishing privileges requires access to local type objects. The object's type-PID is located in the LocalSecurity set, and the type's read- and write-flag obtained there. In the case of a record, field security is obtained from the type object referenced. Maintaining this additional type-PID index avoids accessing the PIDLAM in order to reach the persistent type and discover the type name.

The Offset instruction, which advances to a specific field within a structure, checks component authorisation before extracting its value. The code is generated in such a way that this always jumps directly from an object header to the data required, so that security may be enforced according to information in the header. The Offset instruction includes a parameter giving the size of the value being accessed, so that the corresponding number of read-authorisation bits may be checked. This ensures read protection on the fields of nested records is not lost.

The read-only flag in a header indicates if an the object can be altered. The first step in Committing an object is to consult this flag, and abort immediately if it is set. To ensure read-only fields are not changed without authorisation, the part-protected flag is also checked. If this is set, the type object is obtained and the read-only fields identified. Their values on the heap are compared against the database copy of the object before it is written to the Before Looks File. Any difference causes the Commit to abort.

When an address needs to be allocated to a newly persistent object during Commit, its type is consulted to check write permission. Generally the type will already have been verified against its persistent store counterpart; if not, the definitions and passwords are now compared. If the program did not supply the necessary writekey for the type or any of its components, the Commit aborts.

7.6 CONCLUSION

7.6.1 SUMMARY

This chapter described the P-Pascal security subsystem in terms of granularity, access rights, authorisation mechanism and violation handling. Security has been kept simple, to focus on the practicality of such a feature rather than to design the ultimate security system. Protection is therefore controlled by password specification, and is limited to read- and write-authorisation. Security specification was shown to fit easily into the P-Pascal syntax and the effects of protection were explained. The security information attached to objects and types was described along with its use in establishing and enforcing protection. The additional space requirements are insignificant; but extra local heap dereferences on persistent store retrievals are unavoidable. However system performance is likely to be primarily a function of disk accesses rather than memory lookups. As yet no study has been undertaken to test this hypothesis in P-Pascal.

7.6.2 DISCUSSION

The CPOMS persistence mechanism affected the security subsystem in many ways. The existence of named primary structures enables these objects to have individual protection. In a language where the store comprises environments (eg. Napier) security could similarly be associated with individual bindings. The automatic preservation of data through reachability does not lend itself to access rights on eg. insertion, retrieval, update and deletion, as in conventional database systems where these operations are explicit. This is also true of persistent languages that use other mechanisms such as persistent environments. Automatic retention of data also means that illegal creation of new persistent objects cannot be trapped immediately; only when a Commit (or Stabilise [N88]) occurs. It is more efficient to detect write-violations when a database is updated, rather than on every instruction that changes some value. Therefore a better security system is possible in a language supporting transactions, as such errors can be handled at a logical point in the program execution. Some CPOMS implementation characteristics limit the flexibility of a security subsystem. For example, it is not possible to place confidential parts of a database on a separately protected storage device. In addition, the security associated with a type has to be enforced across the entire persistent store, rather than on a per-database basis. This arises because of the ability to reference data in other databases and the inability to always determine on which database an object will be stored.

Besides the persistence mechanism, the language itself also affects security enforcement. To contrast P-Pascal with PS-Algol for example, the first noticeable difference is that a bulk type has potential for protection that is unsuitable for graph structures. Bulk types lend themselves to access predicate, statistical control and operation-specific security. These are difficult to enforce on a network of objects, as it is relatively easy for situations to arise in which data accessible to a program is not reachable along any authorised path. The first class functions of PS-Algol create opportunities for protection of behavioural information as well, which is not possible in P-Pascal or conventional database systems. The set construct proved useful for security implementation, permitting random access to type objects when type-related security information is needed. A set of strings provided a simple way of specifying an arbitrary number of passwords on Save and Locate.

Both languages already keep type pointers in object headers - for dynamic type-checking in PS-Algol (and Napier88), and for type-clustering in P-Pascal. P-Pascal's nested structures require more space for read-authorisation flags and complicate their inspection. If structures were not nested, a single flag would suffice for any component; P-Pascal can require many bits (depending on the size of the nested object) and must include this size in component access instructions, in order that all bits can be checked. On the other hand, name equivalence facilitates data protection by type. For example, an assignment can only copy an object to another of identical type; hence any read-protection is automatically carried with it. P-Pascal's use of pointers to reference related objects, rather than keys (as in relational systems and DBPL for example), permit a more flexible security system. There can be multiple paths to an object, enabling a program authorised to traverse any one of these paths to use the data concerned. A pointer can be write-protected, which i.a. prevents the particular object it references from being destroyed (made unreachable) - irrespective of type-related authorisations.

CHAPTER 8 SET IMPLEMENTATION

P-Pascal supports persistent sets that can be processed in a non-procedural manner and can comprise elements of any assignable type. The architecture comprises four layers: the Storage Layer, the Primitive Layer, the Relational Layer and the Compilation Layer, as can be seen in figure 8.1. Implementation is constrained by the organisation of the persistent store and the operation of the abstract machine. Indirect addressing is employed by the PPOMS and all database objects share the same files and address space. For example, to reserve disk areas for specific sets, a mechanism other than separate files is needed. Since persistent and transient sets are not distinguished, set manipulation algorithms cannot rely on information regarding the disk organisation of elements. The interpreter works with heap addresses, rather than disk addresses. Thus relationships cannot be computed by address comparison, such as when the same disk block implies the same set, etc. Unlike Codd's relational model, P-Pascal sets need not be in First Normal Form - elements can contain arrays, variant records, pointers and sets. This chapter describes how sets are represented and processed by considering the four layers of the architecture.

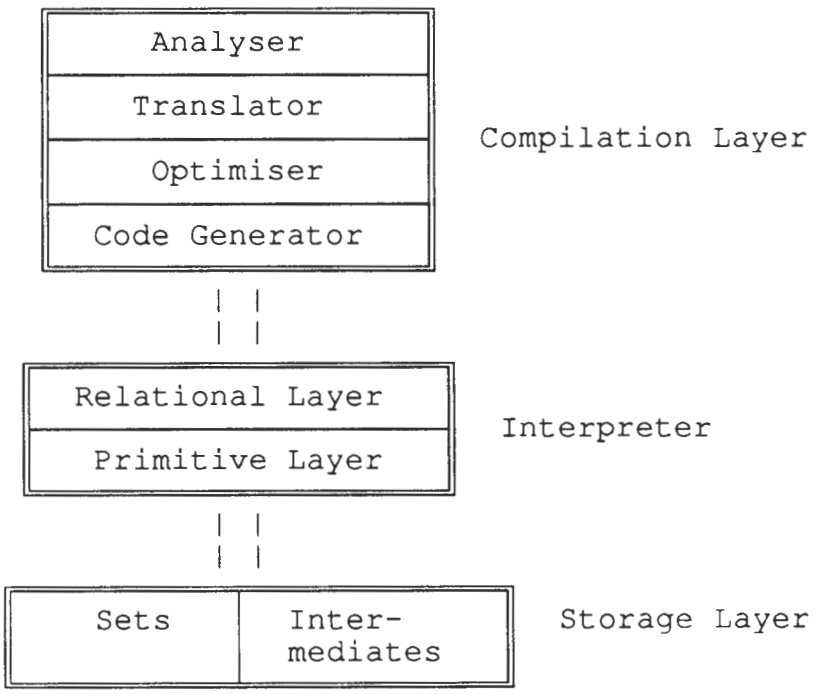
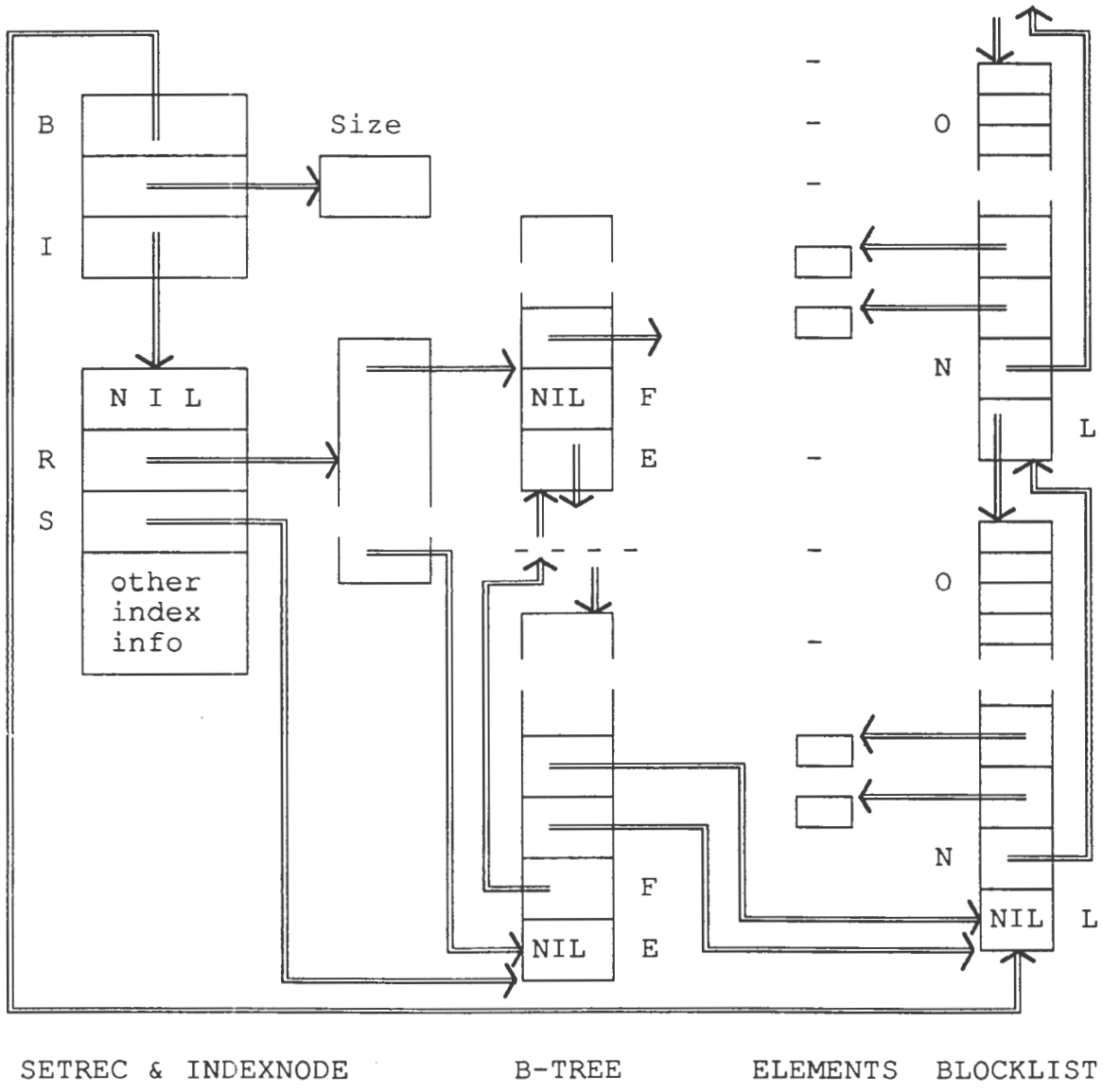


Figure 8.1 Architecture of the Bulk Data Processing Subsystem.

8.1 STORAGE LAYER

All set objects are represented by a record called the "setrec", to which is attached two lists. One is the "blocklist", a doubly-linked list of records called "blockrecs", which contain an array of pointers to the elements, called PtrArr. The other is a list of indexnodes. This section describes set organisation, which is shown in figure 8.2.



B: BlockList; I: IndexList; R: Root Pointer; S: Sequence Set Pointer; E: Previous tip node; F: Next tip node; O: OrderArr; N: Next Blockrec; L: Link to Previous Blockrec.

Figure 8.2. Set representation.

8.1.1 INDEXES

Each indexnode comprises information on one key of the set, including pointers to the corresponding B^+-tree [Baye] root and sequence set (list of tip nodes). One of the indexes is automatically chosen as the prime index and the elements are kept in prime key order in the blocklist. The B^+-tree sequence sets do not contain pointers to set elements. Instead, an entry points to a blockrec and gives the PtrArr component for that element. As will be shown later, this facilitates insertion and deletion as well as other operations, without requiring any additional disk block reads. The blocklist and sequence set are doubly linked to facilitate processing selections of the form [key < N]. In a system permitting concurrent updates of a set, the B^+-tree should be replaced by a B-link tree [Lehm], as is used for example in DBPL [Schm90].

Keys are represented internally by index numbers, derived as follows: Each index is represented by <total-length>, <$offset_1$>, ..., <$offset_N$> where <total-length> is its total size and <$offset_j$> is the offset of the j'th key component in the element. These sequences are sorted into ascending order and the resulting positions are used as the index numbers; this enables set type declarations to list keys in any order. The prime index is index number 1; the <total-length> ensures this is the shortest key.

Programs may declare key components in any order, as these are converted into canonical form, where key components are in order of increasing offset. Where there are two components, both the canonical form key and its reverse are maintained as separate indexes. Thus a key *sno,pno* for set SuppAndParts will result in an *sno,pno* index as well as a *pno,sno* index. This permits operations dealing with *pno* values to be performed as efficiently as those dealing with *sno* values, and incurs no additional cost in terms of duplicate detection. A P-Pascal compiler able to access the persistent store could base its indexing information on the metadata there, rather than on the keys in program declarations. This could build indexes exactly as specified when the set type is first defined: keys *sno,pno* and *pno,sno* would be declared if both indexes were desired.

It is well known that there is no ideal representation for a relation (or set) [Kers]. B^+-trees were selected since hashed storage methods decrease sequential processing performance, and Indexed Sequential schemes were found to be less efficient than B^+-

trees by the Ingres developers [Ston80]. Both these schemes map a number of key values onto the same data page, which must subsequently be searched to obtain the element of interest. In a persistent store with two-level addressing, a set of key values would have to map onto a single PID through which this page could be referenced, since it is not possible to store key value and datafile location in a single indexfile entry.

8.1.2 ELEMENTS

Set elements are separate objects in their own right; as are blockrecs and indexnodes. This allows the interpreter to keep individual elements or nodes in memory when appropriate. It also simplifies set representation and manipulation, as the division of a set into element pointer data followed by element non-pointer data greatly complicates their processing.

A PtrArr does not maintain element pointers in prime order. In this way, no element pointer is moved from the array location where it was initially placed when an insertion or deletion in the PtrArr occurs, and index alteration is avoided. An additional array of one-byte values called OrderArr gives the prime ordering of the elements in a blockrec. Thus to process elements in prime order, a scan proceeds along the blocklist; in each blockrec it iterates over OrderArr to ensure that the PtrArr elements are visited in the correct order.

Alternatives for obtaining elements in prime order were considered. Relational database systems like Ingres [Ston76] keep elements physically in address order; this method cannot be used with sets on the local heap however. Blockrec elements could be sorted in memory when prime order is desired. This is slower than processing elements via the OrderArr, but takes no additional disk space. The prime sequence set could be scanned to obtain prime order. Although requiring no extra space for set representation, more buffer space is used; above all, the additional disk accesses make this method impractical. The PtrArr could be maintained in prime order if indexes only pointed to blockrecs without indicating the specific element. Sequential scanning speeds would improve, but random element retrieval, insertion and deletion would be slower. Indexes could not then be used by the majority of set processing algorithms, which rely on the association of a unique element with each key value.

8.1.3 SET SIZE

Set sizes are maintained so that operands can be ordered in such a way as to diminish element accesses; eg. in nested iterations the smaller set can be processed in the outer loop. Set comparisons are then reduced to a simple integer comparison in many cases: X can only be a subset of Y if its size does not exceed that of Y. A pointer to a set_size object forms part of every setrec. This is preferable to storing the size directly, as the setrec would then comprise pointer and non-pointer data, and would have to be split wherever it occurred as component of another object. The current implementation updates set size with every insertion or deletion; in a concurrent system, this would be done periodically during times of low activity, and would thus represent only an approximation of the current set size: in particular, this could not be used to speed up comparisons.

Copying a set, the equivalent of relation duplication in a conventional database, is an expensive operation. It involves making a new copy of the setrec, blockrecs, indexnodes and elements. If the elements are themselves sets, this is applied recursively. The decision to replicate or not is up to the programmer: set of <entity> or set of <entity pointer> is used accordingly. Temporary sets should not be necessary in P-Pascal, as there are no restrictions on set expression complexity.

8.1.4 OTHER KINDS OF SET

Sets of char, boolean and enumerated types are represented as a bit vector, since their universal set is small. Where the base type of a set is integer, real or pointer, elements are represented as separate objects, with the conventional PPOMS header. This is required for element retrieval and security enforcement, and permits selective retention of elements on the local heap. Elements are maintained in sequence in the blocklist, with no associated indexes. The ordering of elements enables union, difference, intersection, membership testing and set comparison to be more efficiently processed. Sets of integer, real or pointer could have been represented by a B^+-tree rather than a blocklist. This additional space was not considered worthwhile, especially since the majority of accesses to such a set are likely to require sequential scans eg. product, union, difference, intersection, selections like PTRSET [ptrset^.age > 35], etc. When elements comprise a key and one additional field that is a pointer or a simple value, only an index is constructed and not a blocklist. The additional field is instead stored directly in the

sequence set entries. Thus an indexing facility is provided in the guise of a set type. To organise objects into several overlapping data collections, key-pointer sets are used so that the target objects do not have to be replicated. Each kind of set can optionally be packed, as described below. We conclude this section by noting that secondary indexes are not maintained due to the space and time overhead incurred when sets are modified. For similar reasons, a map structure for locating elements of nested sets [Schm90] is not employed.

8.1.5 PPOMS SUPPORT FOR SETS

When a set which is not a bit vector is written to the persistent store, a blockrec and the elements to which it points are stored together on a single datafile block. This reduces the number of disk blocks retrieved when the set is processed. In addition it enables datafile locations, rather than PIDS, to be used in PtrArrs. This has the benefit of halving the number of disk accesses when such pointers are dereferenced: the index file need not be consulted. The size of the PtrArr in a new set is calculated according to the element size. Once blockrecs exist, the PtrArr size is determined from the number_of_pointers in their header. When the number of elements is expected to be small, a PACKED SET can be defined. Such sets have smaller blockrecs and index nodes, and do not have entire disk blocks reserved for them on a database.

Datafile locations cannot be stored directly in a PtrArr, as disk addresses must be negative to be distinct from memory locations. Thus the most significant bit is set, and the datafile location divided by two so that it can be accommodated in the remaining bits. As a result, set elements have an even address; a word is unused where necessary to ensure this. The internal representation of blockrecs required some changes to the Commit, Object Retrieval and Disk Garbage Collection procedures.

When a dereferenced PtrArr value is negative the interpreter converts this value to a datafile address by doubling its absolute value. The LON of the blockrec is located in the Pidlam, and the database number derived from its PID in the usual way. The complete address of the element is thus known, and the object is retrieved into memory. The Pidlam entry for the element contains the object's datafile location and LON exactly as for other objects. The PID field stores the database number and serves to distinguish set element entries from conventional Pidlam entries, as true PIDs cannot be positive.

Commit first checks that all Pidlam objects flagged as "changed" are in a writable database, writing their original contents to the Before Looks File. New descendants of Pidlam objects are given PIDs; then all changed and new data is written to the persistent store. The standard method of address allocation is applied to blockrecs, index objects and pointers within set elements; but only datafile locations are assigned to elements. When objects are written to the persistent store, elements are distinguished because their PID field in the Pidlam is positive. The element is written directly to the given datafile location on the specified database. The prior overwriting of any LONs within the element is the same as for other objects. In the case of a blockrec, any LONs in the PtrArr are overwritten by halving and negating the Datafile-location in the corresponding Pidlam node.

PtrArr entries are associated with fixed Datafile locations so that the allocation of addresses to new elements is simple and fast; that is, PtrArr[i] points to the i'th slot on the block, where a slot is the space occupied by one element. The trade-off is that sets of pointers, integers or reals thus require an OrderArr. Moving PtrArr entries to reflect set order would affect a large number of elements that might not be used by the program. They would have to be brought into memory and flagged as changed, and could not be purged on local heap compaction.

Blockrecs, indexnodes and elements which have been altered are treated identically to any other changed object. When an element is moved because of a blockrec split, it is flagged as changed. Thus its initial contents (and location) are written to the Before Looks File on Commit, and can be restored if necessary. The heap space occupied by deleted elements is reclaimed. If the element exists in the PIDLAM, its LON is negated to indicate that this has become unreachable, as no other object can point to a set element. The local heap space it occupied is thus freed on the next heap garbage collection. A negative LON in the PIDLAM causes Commit to write the object to the Before Looks File (for which purpose its address is retained) and to ignore this object when writing data back to the store. This modification to Commit is well warranted in view of the ability to re-use heap space when the cardinality of a persistent set decreases.

8.1.6 GARBAGE COLLECTION

The marking phase of Disk Garbage Collection marks objects reachable from a (marked) setrec in the conventional manner. During the compaction phase, marked blockrecs and

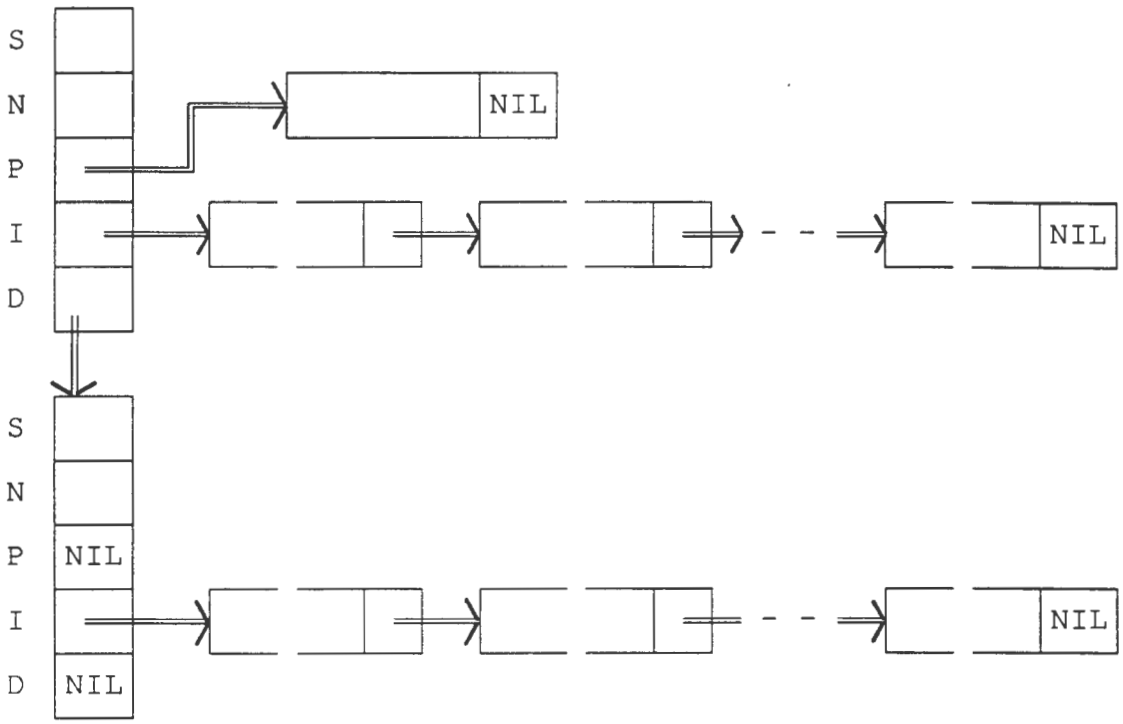
indexnodes are positioned at the next available block boundary, because of their type. Since index pointers do not directly reference elements, only the associated blockrec, these pointers are PIDs and thus remain correct. When a blockrec object is moved, the elements referenced from its PtrArr entries are also moved. Each element is allocated the same relative position within the new block, and PtrArr values are adjusted accordingly. The Garbage Collector should include an optional set reorganisation phase between the reachability and compaction phases, so that elements can be redistributed across blockrecs and indexes adjusted; this feature is not currently implemented.

8.2 INTERMEDIATE REPRESENTATION

In addition to the set objects described above, the interpreter constructs temporary results with more compact representations, according to their use in an expression. The output of a set operation can be one of eight different structures, termed "images". Four kinds of image have been described above: unindexed sets, indexed sets, unindexed but ordered sets and B^+-tree indexes. A set aggregate is represented as an unindexed set. Image types element-marking and index-marking are intermediate results identified by marking elements of a set, or entries of a sequence set, respectively. The remaining kinds of image are referred to as element-lists; these take the form of address-lists or keyed-lists. An address-list is a list of addresses identifying result elements; each "address" in the list can comprise several pointers, if products and joins have been performed. For example, an address-list representing the result of X # Y # Z will comprise three addresses per entry, to identify the three objects that together produce an element of the product. The term "directory" denotes an image which contains, and is sequenced on, some key value: i.e. directories are ordered sets, unindexed-ordered sets, B^+-tree indexes, index-markings or keyed-lists.

An element-list comprises a list of set-descriptors. Each of these is a record from which two chains emanate: the identity-list and a list of projections. An identity-list entry comprises N addresses, optionally followed by a key value. The size of N depends on the number of Joins and Products performed. If an intermediate comprises addresses only and these are PIDs, entries are in address order; where a key is present the entries are always in ascending key order. The projection chain contains the instruction stack location of projections in the expression, so that elements can later be constructed. There is an entry for each operand and intermediate result, in order; or -1 when there is no projection. An intermediate can comprise several set-descriptors, representing the union

of subexpressions of varying constitution - i.e. where the number of addresses or the projection chains differ (see Figure 8.3). The intermediate header includes the number of elements in the set for use in runtime optimisation. The space occupied by intermediate results is freed as it is processed.



S: Size; P: Projectionlist; N: Number of addresses and node kind;
I: Identity List; D: pointer to next set-Descriptor;

Figure 8.3 Intermediate Representation for E1 -> (p) + E2.

Frequently a selection, projection, intersection and/or difference creates a keyed intermediate result to represent a subset of some set S. If the result must be keyed on K, and S has an index on K but the operation is not using this, four alternatives are possible:

(A) Create an address-list of target elements. Then fetch the elements and build a directory on K.

(B) Create an address-list to represent the subset. Then fetch these elements, locating each in the existing K index and flagging the entry there.

(C) Create an address-list of target elements. Sort the existing K index into address order, intersect with the address-list representing the subset and then re-sort into K order.

(D) Create an address-list as intermediate. Then scan the existing index K, each time searching for the corresponding address in the intermediate, and flagging the entry accordingly.

The choice is made according to whether the set is in memory. If it is, the existing K index is flagged to indicate subset inclusion (option B); otherwise a new intermediate is created, leaving the K index of S on disk (option A).

Before describing the layers of the architecture which manipulate images, an overview of the compiler is presented, to outline the strategy decisions that are made statically, and describe the interface presented to the relational layer.

8.3 THE COMPILER

Set expressions are processed by four compiler components. Firstly, the Analyser performs syntactic and semantic checking. Then the Translator converts the expression into an equivalent one which is more efficient to process. The Optimiser decides the processing strategy for the resulting expression. Finally the Code Generator generates PPcode instructions. The Analyser uses standard compiler techniques [Aho] and will not be discussed further.

8.3.1 THE TRANSLATOR

Set expressions are translated into canonical form by the Translator, which also determines element structures and set keys where appropriate. The query optimisations employed in relational database systems are applied [Jark] as well as additional transformations to optimise the PPcode algorithms. A programmer must be aware that transformations occur, and not rely on any specific order of operand evaluation. Transformations are performed statically for efficiency reasons, unlike DBPL where this is done at runtime. The reasons given are that the compiler can then work with a simplified structure and interactive components can then avoid predicate translations [Schm88a].

8.3.1.1 EXPRESSION CONVERSION

Expressions are simplified by eliminating or re-ordering operations and operands, as can be seen from the examples in figure 8.4. Details of specific transformations will be discussed in section 8.5, in conjunction with the algorithms for which they are designed.

```
        [e1,e2,e3] * R - [e1,e5] - [e2]
is translated to
        [e3] * R;

        S [ (a<9) OR (b>9) OR (a>=9) OR (b>12) ]
is translated to
        S [ b>9 ];

        X IN S*Y+Z-W to
is translated to
        (((X IN S) AND (X IN Y)) OR (X IN Z)) AND NOT (X IN W)

        X IN (S # T)   and   X IN (S JOIN T)
are both translated to
        ( X->(S_elm_parts) IN S ) AND ( X->(T_elm_parts) IN T )

        S [p1] * S [p2] - S [p3] + S[p4]
is translated to
        S [ (p1 AND p2) AND (NOT p3) OR p4 ]

        (R # X) + (R # Y)
is translated to
        R # (X + Y)
```

Figure 8.4. Some examples of expression simplification.

Unnecessary intersections, unions, differences, joins and products are eliminated, and ordering is eliminated when used before a product, join, intersection, union, difference, comparison, assignment or another ordering operator. A join of sets that are the same type is converted to an intersection. To reduce cardinality as early as possible, joins are done before Products, projections (and selections) are moved inwards and a sequence of consecutive projections (or selections) are combined into one.

Eg.. (S # P JOIN SP) -> (sno,pno) is converted to
(P JOIN SP JOIN S) -> (sno,pno)

and (S * R) [cost < 10] is converted to
S [cost < 10] * R [cost < 10].

Selections are introduced wherever possible to reduce operand size. Some examples are shown below:

Y[s] * X and Y * X[s] are translated to Y[s] * X[s]

X[s] - Y becomes X[s] - Y[s]

X[f<N] JOIN Y becomes X[f<n] JOIN Y[f<N] if f is a shared field.

There are two exceptions to the general rule moving projections inward. A "fullkey" projection is not moved inside a product. Such a projection includes all components of some key, and thus will not reduce operand size by eliminating duplicate elements. A projection remains fullkey on addition, subtraction or multiplication of key component(s) and independent values. Eg.. if (B.b + 30) is fullkey but A.a and C.c are not, then

(A # B # C) -> (A.a, B.b + 30, C.c) is replaced by

(A->(a) # B # C->(c)) -> (A.a, B.b + 30, C.c).

As the outer projection is performed at the same time as the product, there is no advantage in performing a fullkey projection beforehand.

A projection is not brought inside a difference or intersection. For example

(A * B) -> (a,c,e)

with elements *a,b,c,d,e*, is not simplified to

A->(a,c,e) * B->(a,c,e).

as A and B elements could agree on *a,c,e* values but differ on *b* or *d*. This is true even if the projection is fullkey: it should not be assumed that a key functionally determines a record across multiple sets and it is too costly to enforce this constraint across all instances of a set type. Contrary to the suggestion of Smith and Change [Smit75], a fullkey projection is not moved inward for this reason.

One reason for re-ordering operands is to avoid intermediate result generation; so subexpressions are made the first operand of their parent. Eg.. X # (Y * Z) is converted to (Y * Z) # X. In other cases, such as Joins, operand ordering depends on directory availability. Predicates are transformed into conjunctive normal form, with conjuncts ordered in such as way that indexes may be used to determine candidate elements, then range limits applied and finally conditions on nonkey values checked.

Unlike relational languages, function calls can appear in P-Pascal predicates and projection lists. If the parameters are not taken from the set operand itself, the call is removed and the function invoked once only in advance. This avoids repeated evaluation

and prevents side-effects affecting the selection, desetting or projection. Arithmetic expressions and nested unary set operations are also extracted if they do not reference the cursor, for efficiency reasons. If the current element is a parameter to the function, a list of the candidate elements is made beforehand to prevent side-effects. Eg. S[S.d < F(S.c)] causes a list of S element addresses to be made initially; while S[S.d < F(num)], where num is an independent value, causes F(num) to be evaluated before the selection. Side-effects are also possible in a FOREACH loop, through a function call or assignments to (an alias of) the set being iterated over. An alias arises if two pointers point to the same set occurrence, or a VAR parameter is identical to a variable or another VAR parameter. For this reason an addresslist is always used to represent the control set before iteration commences.

Ingres and System/R apply optimisation rules to decide join order [Seli,Ston76], using simple heuristics to limit the search space - eg. (A JOIN B) JOIN (C JOIN D) is not considered in System/R. The only optimisation on join ordering currently employed by P-Pascal is to precede cartesian products by joins: eg. P JOIN S JOIN SP is replaced by S JOIN SP JOIN P, if P and S have no shared fields.

8.3.1.2 TAILORING THE TREE FOR PPCODE

The Translator modifies or creates projections to obtain canonical form records. In the case of join, it also ensures that duplicate fieldnames are eliminated. Shared fields in a join result are taken from its second operand when a subsequent projection makes this more economical. For example, if pno is a shared field, and jno and jname are fields of PJ elements, then

(SP JOIN PJ) -> (pno, jno, jname)

translates to

(SP JOIN PJ) -> (PJ.pno, PJ.jno, PJ.jname)

rather than

(SP JOIN PJ) -> (SP.pno, PJ.jno, PJ.jname)

so that the result does not require data from SP.

To the interpreter, a join is any theta-join operation, i.e. any cartesian product with a constraint on the element pairs (a P-Pascal join or times followed by a selection). Only a full Cartesian Product is coded as a Product. If Times in the expression tree has a Select

as parent, the Translator thus replaces the two nodes by a join with the corresponding join condition. For example

(S # SP) [S.sno = SP.snum] -> (SP.snum, SP.pno)

is a theta-join and is thus coded as a Join operation, not a product.
During expression translation the equality of shared fields in a join is made explicit in the join condition. For example X JOIN Y where the matching fields are X.a and Y.a, as well as X.b and Y.b for the case where the value of Y.tag1 is 3, is translated to

(X JOIN Y) [(X.a = Y.a) AND ((Y.tag1 <> 3) OR (Y.b = X.b))].

A Select applied to a join causes the predicate to be incorporated in the join condition and the Selection eliminated. Thus (S JOIN SP) [S.city = SP.location] is translated to (S JOIN SP) [S.sno = SP.sno and S.city = SP.location] if sno is the only shared field. In canonical form therefore, a Select is only applied to a variable or aggregate.

A desetting of the form [| predicate |] is replaced by a selection followed by [| TRUE |]. The selection is then handled as described above. In this way the instruction for desetting simply has to return the first element it encounters, i.e. to implement [| TRUE |]. If this is applied to an expression, the Optimiser ensures that this expression is not fully evaluated, but terminates as soon as the first element is found. The Optimiser is described in section 8.3.2.

A set expression is classified as an "indexed set expression" if any of its operands is indexed. Otherwise it is termed "unindexed". Set expressions are further categorised as "simple", "algebraic", "cartesian" or "relational". A simple expression contains a single set variable, possibly followed by unary operators (selection, desetting, ordering and projection). Intersections, unions and differences are algebraic expressions; product and join are cartesian. A relational expression is one returning a boolean result, testing set membership, (in)equality, containment or inclusion.

Four kinds of set assignment instructions exist, so that the ability to copy indexes, rather than rebuild them, can be exploited. Copyset duplicates an entire set structure; this is used for statements of the form L := R. CopyIndexes applies to L := R -> (proj); where the projection includes all and only the fields of R elements so that duplicates cannot occur. This avoids having to reconstruct indexes which are identical to an index of R. Editset occurs with L := L -> (proj) where the projection does not perform any manipulation of prime key fields. Other cases are handled by SetAsg, the default for set assignment.

It was initially envisaged that assignment statements which insert or delete elements from a set be handled identically to those which create a new collection of elements, unrelated to the initial value. These are assignments of simple expressions such as

L := L [s]

and of algebraic expressions such as

L := L * Y - W * (Z + R)

We reasoned that most blockrecs and indexnodes would have to be changed, unless target elements happen to be clustered together. Since all algebraic expression and most simple expression evaluation requires accessing elements, these would be in memory in any event.

However in a persistent environment it is more economical to modify a set than to reconstruct it, to minimise the number of elements, blockrecs and indexnodes that are changed. This increases the amount of space which can be purged on heap garbage collection and diminishes the amount of data that must be written back to the persistent store. For this reason, three kinds of edit instruction exist. In addition to Editset, a simple or algebraic assignment that alters a set is coded as SimpleEdit or AlgebraicEdit, respectively. Since any simple or algebraic assignment can potentially be reduced to set modification in the presence of aliases, SimpleAsg and AlgebraicAsg instructions are distinguished from SetAsg. In canonical form, cartesian expressions contain elements taken from at least two sets; an equijoin of sets of the same type having been replaced by an intersection.

8.3.1.3 ELEMENT AND KEY DETERMINATION

The compiler works with two views of records: it uses the program's notion of field order in its Analysis phase, and the internal canonical form for Code Generation. Element structure must be established for projection, product and join results. The number of fields and their types are easily determined. Their internal ordering is determined from the context in which they appear. When an assignment, a call, an intersection, union, difference or comparison is encountered in a set expression, the canonical form of the destination set, parameter declaration or compatible operand is sent down the tree to associate the correct field order with any derived types below. If these types are used in such a way that no specific canonical form is associated with operand, the convention is to name fields alphabetically in the order they appear in the program code. In such a

case, eg. (X # Y) <= (Z # W), the internal ordering of fields is immaterial, as long as this is consistent across the expression.

It is desirable to establish the keys of an intermediate result in order that duplicate elements may be eliminated as soon as possible. The key of a join or projection is determined from its context where possible. That is, if its parent is an assignment, intersection, union or difference, its keys are identical to those of the other operand(s). To determine the key of a join result when this cannot be deduced from the context, joins are classified into four types on the basis of key overlap. In "same-entity" joins a full key is shared by the operands (eg. PartSet1 JOIN PartSet2). In "entity-association" joins there are shared fields which form a complete key of one operand and part of some key of the other (eg. Parts JOIN SupplierParts) - one operand describes an entity, the other describes some relationship involving that entity. In a "binary-association" the join keys of the two operands have a subset of components that are shared fields (eg. SuppParts JOIN JobParts) - the two operands describe different relationships of the same entity. "Disjoint" joins are those where no shared field occurs in a key of both operands (eg. Suppliers JOIN JobParts) - the relationship between the two operands, if any, is unknown.

Keys for the result are chosen thus: in a same-entity join, any key of either operand can be a key of the result; in an entity-association join, any key of the "relationship operand" can be a key of the result. For binary-association, the result key is formed by concatenating "joinable keys" and eliminating duplicate field names. Keys are "joinable" if they contain the same shared components, and no other key pair has shared components which form a superset of these. Consider X JOIN Y. If X has key *sno,pno* and Y also has key *sno,pno* this is a same-entity join and *sno,pno* is a key of the result. If X has key *sno* and Y has key *sno,pno* this is an entity-association join and *sno,pno* is a key of the join. If X has key *sno,pno* and Y has key *pno,jno* this is a binary-association join and *sno,pno,jno* is a key. If X has key *dno* and Y has key *sno,pno* the join is disjoint and its key is taken as *sno,pno,dno*. With this algorithm the use of superkeys will generally be avoided; but this is not always possible. Suppose we have X JOIN Y with keys *d,b* and *b,c* respectively: we cannot tell if *d* and *c* have the same role, so we assume *d,b,c* is the key of the result. We would be correct in the case of say *Partno,Sno,Jobno* but we'd have a superkey in the case of say *Partno,Sno,Partname* i.e. where *Partno* : *Partname* is one-to-one. If a superkey is used, the result remains accurate but space and time can be wasted.

The key(s) of a "fullkey" projection are the same as those of its operand; the keys of other projections are determined from the program context where possible. For example, in

L := X->(p1)

the key of the projection onto *p1* is deduced from the key of L. This does not mean that values of this key will be unique in X->(p1), but that duplicates must be discarded before assignment. To establish the key of a projection within a product, eg. L := (X->(p1) # Y->(p2))->(p3): the components of the key of L that come from this operand are extracted, and these together make a key for that operand. In the above example, *p3* is compared with the keys of L to obtain a list of keys for the product. Each of these is divided in two: key components that come from X->(p1), and those that come from Y->(p2). These give keys for the corresponding subexpressions. The key of a projection that is the operand of a join (eg. L:= X->(p1) JOIN Y->(p2) ; can only be determined if a key of L comes entirely from that operand. In our example, if a key of L is contained in *p1*, or in *p2*, this is also a key of the corresponding projection.

8.3.2 THE OPTIMISER

8.3.2.1 FUNCTIONS OF THE OPTIMISER

The Optimiser decides the algorithm to employ for each set operator, the scans with which to process its operands and the output it must produce. The output can be one of eight different image kinds, as described earlier: indexed set, unindexed set, unindexed-ordered set, B^+-tree index, set marking, sequence set marking, address-list or keyed-list. The Optimiser chooses the fastest algorithms for the most expensive operations first; then continues to do so for progressively less costly operations within the constraints imposed by these earlier decisions. This is done by traversing the expression tree bottom-up five times, considering a different kind of set operation on each pass. Pipelined operations are also identified during the last pass.

Operators are considered in the following order: (1) join; (2) intersection, union, difference; (3) project, ordering; (4) select, desetting and (5) cartesian product. The number of passes through the tree could be reduced by categorising the operators more coarsely, eg. (2) and (3) could be combined, as could (4) and (5). Times is handled last because it presents no optimisation opportunities. As each algorithm is decided, the scan(s) it will use to traverse is operand(s) and the output format of nested

subexpressions are established. Algorithm selection is based on a node's required output-format (if pre-determined), and the directory(s) available at its children. Recall that a directory is any keyed sequence of element identifications - an index, a marked-index, a keyed-list or an ordered set.

8.3.2.2 ALGORITHM AND DIRECTORY CHOICE

Keys permit more efficient set processing, in two different ways. Firstly, the B^+-tree index can be utilised to rapidly locate elements (in a select or join) or to process elements in key order (which is potentially useful for all operations). Secondly, because a key is a unique identifier, elements which will give a duplicate key value can be passed over before processing. For example, if *sno,pno* is a key for set SP and *sno* is a key for set S, the processing of S := SP->(sno, sname, city) need consider only those SP elements with a unique value for *sno*. These can be efficiently detected because the *sno,pno* index on SP ensures duplicate *sno* values are consecutive.

Where several alternative directories could be used, an ordered list of selection criteria is applied until a unique "best directory" is established. These are:

(1) It is always faster to use a directory where the fields of interest are its most significant components.

(2) If a directory scan and nonkey values are required, and the prime key is a candidate directory, a set scan (blocklist traversal) is fastest. This automatically gives prime order, without having to fetch indexnodes.

(3) The directory used by the parent operation in the expression tree is preferred, since this permits concurrent (or pipelined) processing.

(4) Any directory scan is faster than a set scan, since a directory is shorter.

(5) It is faster to scan an index with a lower index number, as this implies the key size, and thus the sequence set size, is smaller.

Additional rules exist for individual operators, which will be sketched when their algorithms are presented.

8.3.2.3 INTERMEDIATE FORMATS

The result of a set expression is represented as compactly as possible. In the case of unindexed sets, intermediates are represented as address-lists rather than sets. With indexed sets, a result is formatted according to the requirements of the operation applied to it. If the parent operation is ordering, algebraic or relational, the output is represented

as a keyed-list. In the case where the result is a subset of a single set occurrence, an existing sequence set containing these keys (as most significant components) is flagged instead. If the parent is a Product, operand output-format corresponds to that of the Product. Where this is a keyed-list, the key built by an operand comprises those components of the Product output which originate from that set. If an expression is the inner operand of a join then a keyed-list intermediate is created if possible; a complete B^+-tree is necessary in certain situations, to be described later. If it is the outer operand of a join, the output-format is either an address-list or a keyed-list, according to whether or not the keys used by the two operations overlap.

8.3.2.4 PIPELINING INDEXED OPERATIONS

An operation can be executed concurrently with its parent if it performs a single scan through its operand(s) and can process elements in the same order as the parent without loss of efficiency. Where more than one directory is utilised the operation must be executed separately from its parent. A projection applied to an expression is always pipelined with that expression: duplicates are eliminated when its result is created.

A Product can be done concurrently with a parent requiring any key concatenation $K_1\ K_2\ \ldots\ K_n$, where each K_i is (the most significant part of) the directory being scanned by a different Product operand. Operands are re-ordered if possible to permit pipelined processing. Join order is given by the concatenation of (the most significant parts of) the inner join operand's directory and (the most significant parts of) the outer join operand's directory, with duplicate names removed.

Examples:

(SP JOIN PJ) -> (pno) * P2->(pno) + P3->(pno)

If the join is traversing directories *pno,sno* and *pno,jno* respectively, and the Intersection is using index *pno*, then these operations are done concurrently: the concatenation of the most significant parts of the join directory is *pno, pno* which, with duplicates removed, is the Intersection directory *pno*. Both operations are processing elements in *pno* order.

(SP JOIN PJ)->(sno,jno) * X2->(sno,jno) + X3->(sno,jno)

If the join is using directories *pno,sno* and *pno,jno* respectively, and the intersection is using index *sno,jno*, these cannot be done concurrently: the concatenation of the join directories has *pno* as its most significant field and the Intersection index does not. Eg. if

SP comprised [[<P1,S2>] , [<P2,S1>]] and PJ contained [[<P1,J1>] , [<P2,J1>]] then [<S2,J1>] would be processed before [<S1,J1>].

Where desetting is applied to an expression, all its operations are pipelined. This ensures that evaluation stops as soon as an element of this expression is obtained. For example, (X JOIN Y * Z) [|TRUE|] performs the join and intersection together, stopping when an element of the result is found, without computing the entire join and intersection.

8.3.2.5 PIPELINING UNINDEXED OPERATIONS

The first operand of an algebraic or cartesian expression is performed concurrently with its parent. Otherwise, expressions embedded within cartesian products are performed separately in advance, to reduce operand size. A join occurring within an algebraic expression which is not the first operand thereof, is performed separately; other embedded operations are executed concurrently with the algebraic expression. The reasons for this will be given when the evaluation algorithm is described in section 8.5.3. Several alternative ways of processing a selection followed by a projection,
eg. L:=R[s]->(p), were studied:

(A) Using nested loops, perform the selection and projection at the same time.

(B) Select first, marking elements of the source set which satisfy the predicate. Then find the projection over the marked elements.

(C) Select, checking for duplicates on insertion of selected elements into the result [Smit75].

(D) Select first, creating a superset from which the subsequent Project removes duplicates.

(E) Select and create an addresslist result over which the projection is subsequently performed.

The last three options require significantly fewer element accesses; (C) being a little faster than (D) and (E). This ensures that the costly projection performs nested loops over a subset of the operand, without requiring space and time for intermediate creation.

8.3.3 CODE GENERATION

The compiler calculates the maximum number of cursors that can exist at a time in each scope, and allocates a corresponding number of temporary variables. Variables are needed, rather than a stacking mechanism, to cater for situations such as

X [X.a IN Y -> (Y.a + 10 + X.c)]

where the projection references items in both cursors. The position is kept, rather than the contents, as this requires two words irrespective of set type and is needed to obtain the next element when the cursor is advanced. The two words are an object pointer and an offset within this eg. blockrec pointer and PtrArr element offset, indexnode pointer and offset of entry, etc. The Cursor instruction requires the address of the set and of the cursor variable on the stack, for determining where the scan starts. Having the address of the set on the stack also enables aliases to be detected.

The PPcode instructions for manipulating sets are "high-level" instructions in that they take entire set(s) as operands; not individual elements. This simplifies compilation and facilitates P-Pascal enhancement since the compiler is shielded from changes in set representation and low-level execution strategy. It also permits decisions to be made at run time according to set sizes and the presence or absence of data in memory. Low-level code would result in an interpreter with a few simple set manipulation primitives (eg. Next_Element) and a compiler where all the execution strategy is embedded. Object files would be large and code generation would be complex, especially in the presence of pipelining where the steps of one algorithm are interspersed with those of another.

Different set instructions exist for processing indexed and unindexed sets, and for distinguishing pipelined operations (eg. Project, U_Project, Project_P, U_Project_P). Their input and output formats are stacked by separate instructions, along with extra information indicating how operands should be scanned or results created, where appropriate. The list of instructions manipulating sets can be seen in figure 8.5. For brevity, the suite of instructions for the different algorithms are not shown: eg. Join can actually be IndexPair, MergeFetch, TwoPair, SingleIndex or IndexFetch. There is no FOREACH instruction, as this is implemented in a similar way to the FOR loop, except that First and Next instructions are used in place of numeric increments. Pipelined operations take the position(s) of the current element(s), perform the required checks or manipulations and replace this by the position(s) of the next element(s) contributing to the result. They also stack the element constructed in the process, where applicable. In contrast, other set operations replace two set (or image) operands on the stack top by a single set (or image) representing the result. Only a setrec or image header is stacked - the rest of the structure remains on the heap - so the pointer stack is used to ensure that data is not lost in the event of heap garbage collection.

Project	project over an indexed set
Project_P	project over an indexed set, pipelined
U_Project	project over an unindexed set
U_Project_P	project over an unindexed set, pipelined
Fieldspec	projection not requiring duplicate detection
First	get first element
Next	get next element
Get	get element given (partial) key value
Select	select from an indexed set
Select_P	select from an indexed set, pipelined
Join	join of indexed sets
Join_P	join of indexed sets, pipelined
U_Join	unindexed join
U_Join_P	unindexed join, pipelined
Product	find product of indexed sets
Product_P	find product of indexed sets, pipelined
U_Product	find product of unindexed sets
U_Product_P	find product of unindexed sets, pipelined
Intersect	intersect indexed sets
Intersect_P	intersect indexed sets, pipelined
U_Intersect	intersect unindexed sets
Union	union of indexed sets
Union_P	union of indexed sets, pipelined
U_Union	nested union of unindexed sets
SetMinus	difference of indexed sets
SetMinus_P	difference of indexed sets, pipelined
U_SetMinus	difference of unindexed sets
Append	unindexed sets append
SetAsg	indexed set assignment
U_SetAsg	unindexed set assignment
CopySet	indexed set copy
U_CopySet	unindexed set copy
EditSet	indexed set update
U_EditSet	unindexed set update
CopyIndexes	indexed set assignment copying indexes
SimpleAsg	indexed assignment of simple set expression
U_SimpleAsg	unindexed assignment of simple set expression
AlgebraicAsg	indexed assignment of algebraic expression
U_AlgebraicAsg	unindexed assignment of algebraic expression
SimpleEdit	indexed set edit with simple expression
U_SimpleEdit	unindexed set edit with simple expression
AlgebraicEdit	indexed set edit with algebraic expression
U_AlgebraicEdit	unindexed set edit with algebraic expression
IsIn	indexed set membership test
U_IsIn	unindexed set membership test
SetEq,...,SetLT	indexed set comparison
SetEq_P,...,SetLT_P	indexed set comparison, pipelined
U_SetEq,..., U_SetLT	unindexed set comparison
U_SetEq_P,..., U_SetLT_P	unindexed set comparison, pipelined

Figure 8.5 Set instructions in the P-Pascal Abstract Machine.

8.4 THE PRIMITIVE LAYER

This layer acts as an interface between the sets on the heap and the relational operators that manipulate them. Primitive operations which exist for all kinds of image permit insertions and deletions, enable them to be scanned in order (with range limits optionally specifiable), and allow their cardinality to be determined. Other primitive operations are random access (applicable only to a B^+-tree or index-marking) and the sorting, intersection and union of element-lists. This section illustrates the implementation of this layer by presenting some examples of primitive operations.

8.4.1 INSERTION

To insert an element into an indexed set, the prime B^+-tree is used to determine its position in the blocklist. The blockrec of its predecessor in the prime B^+-tree is fetched and the element placed in the first free PtrArr entry. The OrderArr is adjusted accordingly. If the target blockrec is full, this is split into two and the index entries of elements which are moved in this process are adjusted. Instead of a split, elements are redistributed over this blockrec and its predecessor or successor, if possible. Finally the new element is entered in all indexes and the set size is incremented. The address of the affected B^+-tree tip node is recorded each time; if a duplicate is encountered in any index, the process can be rapidly reversed. Insertion into an unindexed set is similar, except that the entire blocklist is first scanned to check for a duplicate. A separate function adds elements without duplicate detection; this is invoked instead when the absence of duplicates is guaranteed. So that this can insert elements as quickly as possible, a new blockrec is always placed at the start of an unindexed, unordered set.

8.4.2 RETRIEVAL

Conventional B^+-tree access routines are employed to locate random elements in indexed sets, given the value of any of their keys. It is also possible to find elements given only the values of the most significant key component(s); if an alternate index is being used and the set is not memory-resident, a candidate address-list is first built before these elements are fetched, to prevent multiple accesses to the same disk block. Functions First and Next traverse the blocklist, recording the current blockrec pointer and PtrArr position and skipping vacated entries. These functions are used for iterating through all kinds of sets and for locating specific elements of unindexed, unordered sets. In an unindexed,

ordered set an element is located by considering the maximum value in each blockrec until the target blockrec is found.

8.4.3 DELETION

In an indexed set, B^+-tree deletion is used to determine the blockrec and PtrArr entry referencing the target element, and to remove the element from all indexes. The corresponding PtrArr entry is set to null in the target blockrec, and the OrderArr adjusted. This blockrec is not merged with its predecessor or successor because of the time taken to alter blocklist and indexes, and also because several elements and index nodes could be changed, and need to be retained in memory until Commit. Finally the set size is decremented. If the desired element was not present in the set, no error is raised since a statement like X := X - [E] is correct if X does not contain E.

8.5 THE RELATIONAL LAYER

8.5.1 OVERVIEW

Set expressions are executed in such a way as to reduce operand sizes as much as possible, as early as possible; and to make the maximum use of directories [Smit75]. Consecutive operations are performed concurrently, or "pipelined", wherever this can be done without decreasing performance. Intermediate results are represented as compactly as possible and algorithms are designed so that each disk block on which a persistent set resides is accessed once only when the set is processed. For example, if an operation requires using an alternate index and then fetching elements, an ordered addresslist of target elements is created first. All functions free up or re-use heap space taken by temporaries as soon as possible.

The abstract machine comprises an Execution_Loop which is performed repeatedly until program termination. This calls an Interpret function to invoke the routine corresponding to the current instruction. Each set operation is executed by a single function. When embedded instructions are encountered - for example in the predicate of a join or a select, or in a projection list - the routine calls the Interpret procedure to process this, without returning control to the Execution_Loop until the entire operation is complete. The data stack is used to store information such as the start address of the instruction (to permit

iteration through the instruction sequence for every element) and the current element; the pointer stack stores the heap location of the result.

An operation which must discard duplicates identifies these before processing wherever possible. This occurs when duplicates can be detected from the most significant components of the directory being traversed. As an example, consider (SP JOIN SJ) -> (sno) using indexes *sno,pno* and *sno,jno*. After an element is placed in the result, the join algorithm is not applied to successor elements until the *sno* value changes.

8.5.2 INDEXED SET EXPRESSIONS

8.5.2.1 SIMPLE EXPRESSIONS

Set expression evaluation is pipelined with a subsequent assignment operation, so that the space occupied by the destination set can be re-used as result elements are generated. Conventionally an interpreter stacks a value and subsequently assigns it to a destination address; with set objects these two steps are performed by a single instruction so that the heap space can be re-used directly. Otherwise time and space is taken up by creating the set on free heap space, and then copying the data from there to the original blockrecs, indexnodes and elements. Since only the "setrec" itself may be referenced by other objects, there is no possibility of subsequently dereferencing an indexnode, blockrec or element PID from within another object. The alternative of freeing the original space can require changing the PIDLAM entries of all these objects, to reflect their new LON. Another advantage of pipelining evaluation and assignment is that knowledge of the destination and source permits aliases to be distinguished immediately; when this occurs, the set is modified rather than re-created, to reduce the number of changed objects. In limiting the number of objects changed when a set is altered, sparsely occupied blockrecs can occur, since these are not merged when elements are deleted. Set compaction is best performed on Commit, alternatively this can be done by means of a separate utility program or on garbage collection of the persistent store.

In an earlier version of the system, algebraic expressions were not pipelined with a subsequent SetAsg. A method was developed for dynamically distinguishing insertions and deletions, to prevent reconstructing entire sets when simply adding or removing a few elements. This method treated set aggregates differently from other set expressions, in that where X and Y were aliases of L

L := X * Y + [e_1, e_2, ..., e_N]

would cause insertion of e_1,...,e_N but a new set would be constructed for

L := X * Y + Z

The Setasg instruction first creates the prime index of a set, to obtain the correct element order; its sequence set temporarily points directly to the elements. Thereafter this sequence set is traversed and the blocklist constructed. During this process, the prime sequence set is changed to reference blockrec and PtrArr, and alternate indexes are built. If the elements are presented in prime order, the two steps are combined. Only a percentage of each blockrec is filled, as indicated by the database "load factor", to leave room for subsequent insertions. CopySet and CopyIndexes make use of an oldblockrec : newblockrec mapping so that values in tip nodes can be adjusted when B^+-trees are copied.

If elements contain sets themselves, the set assignment operations recurse. Setrecs embedded within structured objects are distinguished from other components by being positioned at the start of the data, as explained in chapter 4. This is necessary because they require different action on assignment (the blocklist, indexes and elements must be assigned) and on equality tests. All structure assignments check for the existence of such components, to ensure that entire sets are assigned, and not just the setrecs. The information on embedded sets is encoded within the objects themselves, because sets can be nested to any depth. For example, if separate instructions were being generated for set components, a record assignment such as R1 := R2, would require separate instructions for set and non-set components at all levels of nesting.

Selections and projections in a simple statement are always pipelined with SetAsg. If the source set is not in memory, an assignment traverses this in address order, rather than in the order required by the prime key of the destination. For example, in

S := SP -> (sno,sname,city)

where the prime key of SP is *sno,pno* and that of S is *sname* the *sno,pno* index is traversed to detect duplicates; although the other index would give elements in the correct order for S, this could cause SP blocks to be retrieved more than once. An *sname,pno* index on SP is used instead if this set is in memory. As the implementation of ordering and desetting is straight forward, this section describes only the selection and projection procedures.

8.5.2.1.1 SELECT

Database systems like Ingres and System/R process selections by means of a single index (or set) scan, chosen according to the given predicate and the set organisation [Ston76,Seli]. System/R uses a complex decision process, which is necessitated in part by its facility for distributing a single set over a file, with very few elements on the same page. It also takes into account the input order required by the next operation, when this is a join, ordering or GROUP BY [Seli], and uses statistics on set cardinality, the number of pages occupied by the set and each index, the number of distinct key values in an index, etc.

In contrast, P-Pascal uses several indexes where possible, to reduce the search space and minimise the total number of accesses. In Canonical Form a Select can only be applied to a set variable, not an expression. The predicate is in Conjunctive Normal Form, with conjuncts ordered as follows: key comparisons occur first, then key disjunctions, then range comparisons and finally the remaining nonkey conjuncts. A key comparison is a conjunct of the form *expression = keyvalue* , where *keyvalue* is (the most significant component of) some index. A key disjunction is one comprising only key comparisons. A range comparison has the form *expression op primevalue*, where *op* is the operator <, >, >= or <= and *primevalue* is (the most significant part of) the prime key. Selection uses indexes to obtain a list of candidate elements fitting each key comparison; their union (intersection) is taken for disjunctions (conjunctions). Lists include output components obtained from indexes where appropriate; these are consolidated as the lists are combined. Thereafter, conjunct(s) limiting the range of prime values cause entries out of range to be removed. The remaining elements are fetched to see if they satisfy the nonkey conditions, if any, or to obtain the desired output format, if necessary. This approach is similar to that of SQUIRAL [Smit75], extended to handle expressions rather than constants in key comparisons. Any expression can occur in a key or range comparison, unless it includes a reference to the cursor element, eg. in

S [(S.b = F(S.c) AND (S.a = 3 * Sum)]

the first is not a key comparison, but the second one is, if Sum is not a field of S elements. Multiple indexes are not used in System/R and Ingres because this requires locking relations while the intersections and unions are performed.

8.5.2.1.2 PROJECT

In the absence of a directory, projection is typically performed using nested loops through a set (marking duplicate elements), by sorting the set, or by building a hashed-table or directory [Date]. If the most significant component(s) of some directory constitute a key of the projection, the result is determined by skipping elements until the key value changes. In addition to the general rules for directory selection, where a projection can use either of two directories to detect duplicates, one which includes the remaining items in the projection list is employed if possible, to avoid element access.

Where there is no directory from which to identify duplicates directly, P-Pascal either uses nested loops (marking duplicates), or traverses the set once, detecting duplicates on insertion into the result; the overall cost is approximately the same. Where the output must be a keyed-list, the latter method is required. If two projections over the same set are being pipelined, as in X->(p) * Y <= X->(p) - Z, marking cannot be used for both as only setrecs are stacked, not entire sets. Thus there is only one copy of each X element (on the heap) that can be marked. Sorting is not used in P-Pascal; besides being expensive in terms of space and CPU time, this method is not applicable to projection lists containing sets, as these cannot be ordered.

8.5.2.2 ALGEBRAIC EXPRESSIONS

P-Pascal implements the algorithms proposed by Smith and Chang [Smit75] to process the intersection, union and difference of keyed sets in such a way that no disk block is retrieved more than once. These algorithms also have the property that consecutive intersections, unions and differences can be pipelined when the order of processing operands is the same. The algorithms require access to both sets in the same order: using any directory common to both, and a set scan instead where this is the prime index of the operand. If there is no common directory, a directory is built for one of the sets, to match the physical order of the other set. Set sorting is not employed, as this is not possible when the elements themselves contain sets. If the algebraic expression is the operand of a join, it uses the join index if it can so that the operations can be done concurrently.

The algorithm [Smit75] proceeds as follows. The two sets are merged initially (traversed in the same order) and the elements potentially in their intersection are memoised in a set I. Thereafter, the operand(s) which were accessed through a directory are traversed in a

single set scan, consulting set I to identify duplicates. Thus (A + B) is given by { A, B - I}; (A * B) by I, and (A - B) by A - I.

When the output-format is an intermediate result, the candidate element is chosen from the simplest subexpression, to facilitate subsequent materialisation. Elements are taken from the operand with the least number of addresses (i.e. derived from the fewest number of products and joins). Otherwise an operand having no associated projection is preferred. If both sets are single-address, no-projection operands then result elements are chosen from an operand where the output-format corresponds to prime order, so that each block is accessed once only.

8.5.2.3 CARTESIAN PRODUCT EXPRESSIONS

8.5.2.3.1 PRODUCT

Products are computed by nested loops; the type of scan is determined from the Product's output-format. If this is a set, operand set scans are used. If this is a keyed-list each operand traversal is suited to the output components which it must provide: if these do not occur together in any one directory, a set scan is needed.

8.5.2.3.2 JOIN

The join methods commonly employed by relational database systems are nested loops, nested loops using a directory on the inner operand, sort/merge (whereby the relation is sorted on the join attribute(s) if necessary and the operands are processed in a merging scan) and hashed-table joins [Seli,Date]. Algorithms that make use of links (pointers) between elements of different relations were developed for System/R [Astr76]. An evaluation of the four most common join methods in [Blas] showed that a merge on clustering indexes (prime indexes in P-Pascal) is uniformly the best method; otherwise cost estimates need to be made in terms of the data properties in order to choose the optimum join algorithm for a particular situation.

In the sequel, "joinfield group" refers to the fields of an operand appearing together in a single join comparison. For example in

WHERE (X.a + X.b * X.c < Y.num) AND ...

the joinfield groups are *a,b,c* and *num* respectively.

There are five join algorithms in P-Pascal. An *IndexPair* join scans two directories in a merging scan. A *MergeFetch* join follows this up by accessing the element pairs, to check remaining join conditions. To ensure no block is accessed more than once, the list is ordered on the addresses taken from the first operand. The first set is then scanned to retrieve the elements in the list, and create another list comprising address-subelement pairs, ordered on address. The subelement comprises values of the first operand required for testing the join predicate or for the join output; the addresses identify elements of the other operand. These elements are fetched in the final step and the result created. The first list is not needed if the outer operand is being traversed in a set scan.

A *TwoPair* join performs a merging scan of two direcory pairs and intersects the resulting address-pair lists. Since a real world object often has two identifiers (eg. Number and Name), there are many situations where exactly two joinfields exist and both are key components. For the same reason, it is not worthwhile using two directory-pairs if set elements have to be fetched eventually; as the two keys are likely to be in a one-to-one relationship, the use of a second directory pair will generally fail to reduce the search space any further. If target elements have to be fetched to obtain output fields or because of a projection applied to the join, the *TwoPair* join is replaced by a *MergeFetch* join. A *SingleIndex* join performs a set scan through the outer operand, using a directory of the inner operand to identify elements having desired joinfield values. An *IndexFetch* join follows this up by fetching inner operand elements to check remaining join conditions. An address-ordered intermediate is created first, so the inner set can be processed block by block. Each entry contains the joinfields and output components of the outer operand element and the address of the corresponding inner operand element. If no directory on any joinfield exists in either operand, a temporary B^+-tree is built for the inner operand, containing all joinfields and join output values. If the inner operand is a product or join that can only output some of these fields, the remainder are excluded from the output to avoid retarding the sub-expression evaluation.

If the outer operand of a join is an expression, its result is either an address-list or a keyed-list, according to the directories being used. Suppose the inner operand of the join has directory(s) on joinfields J_1 ... J_n. The outer operand's output is an address-list unless it uses some of these J_i's. That is, if it is using a directory containing one or more J_i's, or is fetching set elements, then its output-format is a keyed-list. We shall call those J_i's which are being used by the embedded operation U_1, U_2, ... U_m. The key built depends

on the possible directories that the other join operand can use. Suppose these directories are I_1, ..., I_k (each contains one or more of the joinfields J_1, ..., J_n). One of these which includes U_i's is chosen, the outer operand creates a keyed-list comprising these U_i's and the join is *MergeFetch* (or *IndexPair* if these U_i's incorporate all joinfields). The prime key of the inner operand is the I_i directory chosen; otherwise that which includes the most U_i's is employed.

Consider X[s] JOIN Y with joinfields A,B,C such that Y has two indexes A and B,C. Here the J_i's are A, B and C.
The I_i's are: I_1: A and I_2: B,C.

(1) If the Select is using indexes D,E and F,G on X, without fetching elements, then its output must be an addresslist, as D,E,F,G doesn't contain any of A,B,C.

(2) If the Select is fetching X elements, then it creates a keyed-list on A,B,C.

(3) If the Select is using a B,E directory on X without fetching elements, then B is U_1, i.e. the only field being used by the Select which is also a J_i. So the Select's output-format is a keyed-list on B.

The Optimiser selects the most efficient join strategy possible, according to directory availability. The order of preference is: IndexPair, TwoPair, MergeFetch, SingleIndex or, at worst, IndexFetch. To minimise the work involved when an IndexFetch join is necessitated, operands are examined by the compiler for possible re-ordering. It adjusts the projection following the join accordingly, if re-ordering is warranted. This occurs when:

. both operands are set variables, and only the outer one has an index that can be used by the join;

. the inner operand is an embedded operation and the outer one is a set variable with an index that can be used by the join;

. the outer operand is an embedded operation which is processing data in order of some joinfield group, and the other operand is a set variable with no index for joining;

. both operands are embedded operations, and only the outer operand is processing data in joinfield order;

. both operands are expressions neither of which is processing any joinfield group, and the inner one is the more expensive operation;

. neither operand has a directory for joining, but the outer operand has a directory containing all joinfields and output values amongst its less significant components (so a set scan can be avoided).

Directories are used to resolve any comparison in the join predicate. Where there is a choice of directory to use in a MergeFetch, SingleIndex or IndexFetch join, a directory is chosen according to the characteristics of the comparison where its component(s) occur. An equality is preferable to a condition having some other relational operator; and a comparison with the fewest number of components in the joinfield groups is used. Non-equijoins (i.e. a restricted cartesian product where the only constraint is not based on equality) also cause operands to be re-ordered by the compiler if necessary. For example, consider (X # Y) [X.num * X.amt > Y.val] with indexes on *num,amt* and on *val* respectively. For each *num,amt* pair, the Y index is scanned only until *val* becomes too large. This should be contrasted with (Y # X) [X.num * X.amt > Y.val]. The nested loops through the *val* and *num,amt* indexes cannot be terminated prematurely, as *amt* is not steadily increasing in its index. Hence this is translated into the earlier expression. Although iterations with Y in the outer loop can be terminated early when *amt* is a subrange of the positive integers, the evaluation of (X # Y) [X.num * X.amt > Y.val] is still more efficient than computing (Y # X) [X.num * X.amt > Y.val].

8.5.2.4 RELATIONAL EXPRESSIONS

A single function Is_Subset is used by all six relational operators:

A < B is Is_Subset(A,B,True) {True implies proper subset}
A <= B is Is_Subset(A,B,False)
A > B is Is_Subset(B,A,True) i.e. B < A
A >= B is Is_Subset(B,A,False) i.e. B <= A
A = B is Size(A) = Size(B) AND Is_Subset(A,B,False)
{ i.e. same size and each element of A is in B }
A <> B is (Size(A) <> Size(B)) OR (NOT Is_Subset(A,B,False))
{i.e. different size or some element of A is not in B}

Is_Subset first compares set addresses and then set sizes. If this is not conclusive, it compares the operands using a variation of the intersection algorithm, which terminates prematurely if a key from the first operand is found to be missing from the second. The

only difference between inclusion and proper inclusion is that the latter checks that the first operand has strictly fewer elements than the second operand. Since the location of sets within structures can be determined from object headers, set equality is used accordingly, in place of simply comparing the entire contents of two objects. In canonical form, IN can only be applied to a set variable, aggregate or projection. If the set operand is a variable, any index on the set can be used to find the element. A Fieldspec instruction is generated so duplicate detection does not take place. As aggregates are represented as unindexed sets, a set scan is required.

8.5.3 OTHER SET TYPES

The processing of sets represented as bit vectors is well known; the introduction of selection, desetting, ordering and product is easily managed. The algorithms manipulating unindexed sets are similar to their indexed counterparts, with two exceptions. Firstly, it is not possible to build a join index when a shared field is a set, since sets are unordered. For this reason, a sequence of such joins and products is pipelined together (with joins preceeding products as explained earlier). A more marked difference exists with algebraic expression evaluation. Directories cannot be built to detect common elements, since the sets have no known keys; and sorting is not possible in cases where elements themselves have set components.

A sequence of unindexed intersection, union and difference operations is performed concurrently (pipelined). The first operand is called the generator; the others are termed comparators. This reflects the algorithm, which evaluates the expression by considering each generator element in turn, using the rest of the expression to determine whether this belongs in the result or not. Short-circuit evaluation of expressions is used. For example, S - (X * Y + W * Z) has a JumpIfTrue after the first intersection. Unions which are not nested within an intersection or difference are processed separately using an Append instruction. Eg..

L := R - (X * (W + Y)) + (Z * V) + T - S

is treated as

L := R - (X * (W + Y)) - S; L := L + (Z * V) + T;

The compiler translates an expression such as (X + Y) * Z to Z * (X + Y) so that an Append is avoided, since it is a more costly operation.

The instruction stack is traversed for each generator element in turn. For each operand encountered, a boolean is placed on the stack to indicate whether the current element occurs in this operand. For each operation, the data stack is popped and replaced by the result of the operator application: intersection, union and difference reduce to AND, OR, and AND NOT, respectively. This not only saves intermediate creation and accessing, but also permits short-circuiting to reduce iterations.

Where the generator is a projection on some aggregate or variable, the projection could be applied in advance, but it is preferable to pipeline operations wherever possible. It is inefficient to apply the algorithm to all generator elements, and detect duplicates on insertion into the result. The alternative of checking each element against the result before applying the algorithm to it, is also too slow. The system thus pipelines the operations, detecting duplicates by marking them in the generator, and ignores marked elements when the next element is required. If the algebraic expression is pipelined with any another operation, the compiler ensures that marking sets of this type does not occur there. The distinction is made on the basis of set type because of the possibility of aliases.

If a comparator is identical to the generator, the value True is stacked immediately. Similarly, if a set occurs more than once as comparator, it is not searched multiple times. Where a comparator is a pipelined selection, the operand is not searched if the predicate is false for the current element. Thus a statement such as

```
L := X - (Z + Y[s])
```

is converted to

```
Temp := Y[s]; {reduce operand size beforehand}
L := X - (Z + Temp[s])
```

so that Temp is not searched in vain when the generator does not satisfy the predicate s.

To determine whether the current element occurs in a product it is divided into sub-elements and then each operand is searched independently. Thus if the comparator is (X # Y) -> (X.b, Y.d, X.f) then firstly X is scanned to find an element with the desired values for X.b and X.f; if successful, Y is searched for an element with the desired Y.d value.

Any comparator that is a projection is handled in a similar way - instead of performing the projection in advance, the operand is searched for an element comprising the desired

field values. Whether this is cheaper than computing the projection in advance, depends on the relative sizes of the generator and comparator, and the selectivity of the projection. The pipelined approach was adopted as it saves intermediate result space. Element marking cannot speed up the evaluation as the elements being considered are under the control of the generator.

A join occurring as operand in an algebraic expression is performed separately in advance, to reduce operand size. If this were not so, in an expression like

(X JOIN Y) * (W JOIN Z)

where each element in (X JOIN Y) matches an element of W, W JOIN Z can be evaluated many times over. Iterations can only be terminated prematurely if elements of W and Z are encountered which are joinable and together match the (X JOIN Y) element currently being sought. The relative costs of performing the join before and during the algebraic expression depends on the relative sizes of the generator and the join operands. The larger the generator set, the more advantageous it is to perform embedded joins initially.

The generator is always the first operand. For example, in L := X - W * V; X is the generator because result elements all originate from X. A statement such as

L := [e1,e2,e3,e4] * R is translated to

L := R * [e1,e2,e3,e4];

unless there is only one element in the aggregate, or the other operand is an expression, so that an element of R is scanned once only. Eg..

(R # X JOIN Y)->(p) * [e1,e2,e3,e4];

is converted to

[e1,e2,e3,e4] * (R # X JOIN Y)->(p);

Other transformations that reduce the number of iterations performed are illustrated by examples:

(1) Y[s] * X[s] replaces Y[s] * X or Y * X[s];

(2) Y->(p) * X replaces X * Y->(p);

(3) (Z JOIN Y) ->(p) * X replaces X * (Z JOIN Y) ->(p);

(4) W * (X # Y) replaces (X # Y) * W.

Unnested unions are executed by the Append instruction. Several expressions can be appended to a set with a single instruction; this enables the sets to be considered in increasing order of size in order to reduce the total number of iterations. At the outset,

each set is compared against the destination, as well as against the others to be appended, to detect aliases.

8.6 CONCLUSION

8.6.1 DISCUSSION

This chapter described the different internal set representations used in P-Pascal. PPOMS support for sets, in terms of storage, alteration, retrieval and garbage compaction was outlined. The overall strategy for processing set expressions was presented and the four layers of the architecture described.

We have shown how the integration of data type completeness and orthogonal persistence with the Pascal set construct provides a simple but powerful way of supporting data collections. This single constructor translates into an indexing mechanism, an ordered or unordered collection of objects, with any number of B^+-tree indexes to accelerate element processing. Each of these alternatives can optionally be PACKED to reduce the space occupied; if not, the data is organised physically on disk blocks in a way that diminishes access times. Elements can contain any kind of data; in particular, variant records accommodate non-homogeneous data collections.

The persistent store organisation constrained the possible set representations, but implementation goals were achieved without having to change the persistence mechanism. Direct addressing of elements was possible because no data object outside a set can point to any element. The application of this addressing mechanism to other internal pointers is possible, but has not been implemented. The inability to use separate files for different sets has only a minor impact on processing speeds when blocks are reserved for individual sets.

Since transient and persistent sets are indistinguishable to the interpreter, data relationships cannot be deduced from disk addresses, as in conventional database systems. An initial implementation mapped all sets onto disk locations - irrespective of their longevity. The final design was able to dispense with this approach and retain the CPOMS persistence mechanism. The requirement that pointers be separate from non-pointer values had no major impact on set representation, since it is in any case unreasonable to store a bulk object as a single entity. We conclude by noting that the

CPOMS characteristics mentioned here are retained in the Napier stable store, so the comments apply to that system as well [Brow89].

The integration of a relational-like language and a programming language affords several opportunities for optimisation which are not feasible in an environment with two separate languages handled by two independent systems. Global optimisation is possible, and should have a considerable effect on performance. This has been exploited in P-Pascal by extracting embedded operations and function calls from set iteration, projection, desetting and selection where possible. Procedure analysis can be used to determine which intermediate results and temporary directories are useful, when they should be constructed and how long they should be retained. Other advantages over relational systems permit expressions to be simplified according to the program context. For example, when a derived set is being assigned to a keyed set variable, these uniqueness constraints on the destination elements can be used to eliminate duplicates at an early stage.

P-Pascal sets are not restricted to first normal form relations, and support a richer collection of element types. Work on extending the relational model is therefore applicable here [Osbo,Schm85,Ston83]. P-Pascal databases are effectively a mix of relational and network database structures. Programming methodologies and database design strategies in such environments deserve some attention.

As a result of the implementation effort, it appears that building a bulk type into a language is preferable to having programmers develop their own data representation and manipulation strategies for such data. Programs using high-level operators are more concise, and easier to understand, maintain and reason about. The bulk type and operators have been introduced into Pascal without undue complexity of notation or semantics. The complexity of the implementation is greatly increased; however this would appear to be an argument in favour of building-in bulk types: there are so many ways in which set manipulations can be improved that it requires considerable effort for programmers to produce equally efficient programs in the absence of a bulk data type. Naturally, the provision of such a type does not preclude programmers with a different bias from creating their own data structures and functions for data collections - this is one advantage of a persistent language over a relational database system.

We conclude by evaluating the P-Pascal implementation in terms of the requirements for bulk data types laid down in [Atki89]. Efficient set representation has been achieved by allocating elements to disk blocks in such a way that accesses are minimised: indexnodes are of block size since this minimises the number of levels in the B^+-tree and brings a maximum of index information into a buffer at a time; blockrecs and their elements are physically stored together and elements are in prime order. Datafile locations are used to access elements directly. Persistence is efficiently supported and sets are compact, clustered and indexed as suggested [Atki89]. Their operations have been designed to maximise efficiency by compiler translation of expressions into efficient canonical form, by pipelining, by avoiding repeated fetches of a block, by reducing set sizes and utilising directories as much as possible, etc. The issues of concurrent and distributed access to sets have not been addressed.

8.6.2 APPLICABILITY TO OTHER LANGUAGES

A corresponding type constructor can be incorporated in PS-Algol with a minimum of change to the underlying implementation. The main difference is that elements with structured components use pointers to separate objects. This complicates the handling of such element components for directory creation, element comparison, predicate testing and value extraction (eg. in a projection). It would appear that concepts like keys, ordering and selection or desetting predicates are inapplicable to sets of functions. The meaning of these constructs in relation to graphics types needs to be defined. The other set operators can easily be specified for sets of any kind.

The observations above apply equally to the introduction of sets in Napier88. Of particular interest however are the new data types: universally quantified procedures, existentially quantified abstract types, environments and the type ANY. Polymorphic procedures would be useful for providing aggregate functions, eg. finding the maximum value in a set or producing a set of [<value, aggregate>] records. An example of the latter is GROUP_AVG(SP,sno,cost) which produces records comprising supplier number and their average cost in SP. Environments are constructed within the Napier system; they are not directly supported by the PAM machine. A bulk data type which provides an indexing mechanism, such as the index-only sets of P-Pascal, would be a convenient way of implementing the string-to-object mapping that is an environment. Objects of type ANY are represented as pointers to heap objects. As such it would be as easy to inject sets into ANY and project them out again, as for any other type of object. In other

respects, the effect of type ANY is the extra level of indirection, which retards set manipulation.

The Napier88 machine supports objects that comprise at most two words on the main stack or two pointer words. Thus a set object would have to comprise at most a blocklist pointer and indexlist pointer; the size would have to be discarded, or incorporated in an indexnode. Any alternative would require an extra level of indirection to reach set elements, which should be avoided. The Napier abstract machine represents structured data as a pointer to a heap object. This enables polymorphism to be supported using a partly-tagged architecture [Conn90]. Objects whose type is not known statically are padded to occupy four words (two scalar, two pointer) and interpreted dynamically using a tag stored with the object. In such an architecture, sets could be handled in the same way as any other type, occupying two pointer words (and two unused scalar words when the type is abstracted over). The P-Pascal approach of nesting structured objects would require a different mechanism for implementing polymorphism. Either textual polymorphism as in Ada [Ichb] or a uniform implementation of types as in [Car83] could be used. The trade-offs associated with these methods are discussed in [Conn90].

CHAPTER 9 CONCLUSION

9.1 SUMMARY

This thesis has described the P-Pascal language and the abstract machine that supports it. It showed how the CPOMS could be extended to provide for clustering, metadata management, security and bulk data types. This chapter reviews the thesis and makes suggestions for future research.

Chapter 2 provided a background on database programming languages in general, and persistent languages in particular, by discussing the most well-known examples. Design criteria for persistent languages were proposed. Chapter 3 introduced the P-Pascal language, focusing on the treatment of type complete sets. This single type constructor can be used to represent a wide range of data collections of differing characteristics. The bulk data type has been incorporated into a conventional language without introducing any new types, in accordance with the original goal for persistent language design, as set out in [Atki83]. Differences between P-Pascal and PS-Algol, Napier, DBPL, Pascal/R and Galileo were discussed.

Chapter 4 sketched the P-Pascal abstract machine and highlighted areas where a different approach from that of PS-Algol was taken, to investigate the flexibility of the CPOMS to handle different languages and data representations. The tradeoffs of nesting structures or using pointers were discussed. The system was shown to be capable of supporting complex objects without substantially increasing the complexity of the interpreter.

Chapter 5 described a data clustering mechanism for a persistent store, based on the notions of type-clustering and family-clustering. The persistent store organisation was evaluated in terms of its potential for object placement strategies. Clustering algorithms must be kept simple and efficient, as they need to be applied on program Commits as well as on datafile compaction, which are both complex processes. The indexfile provides a great deal of scope for a physical placement strategy, since 1K of these 32-bit entries are retrieved from disk at a time.

Chapter 6 discussed the implementation of an automatic metadata maintenance facility and justified the choice of this approach over the conventional method of using a schema. Information collection is a service to programmers; there is no restriction on the data

types allowed on the store, and new types do not have to be explicitly added to a database. The metadata subsystem highlighted the need for concurrency. A single "data dictionary" for the entire store is essential because inter-database references are permitted, but locking is only supported at the database level. The P-Pascal language facilitates type management in several ways. Eager type-checking is possible because pointers are typed, so dynamic checking is only necessary when a new primary (named) object is Located. Sets are useful for storing and manipulating metadata, particularly when rapid random access to type objects is required. On the other hand, the security subsystem complicates the processing of types: authorisations must be established according to program scope.

In chapter 7 the consequences of enforcing a simple security system in a persistent language were discussed. The cost of establishing type-related protection on every database retrieval is unavoidable. Since authorisation data is kept in memory, we postulate that this should not have a marked effect on the speed of database-intensive applications. Aspects of the persistent store design affecting security include the inability to keep highly confidential database items on separate files, the need for authorisations to apply to the entire store because of inter-database references, the ability to access an object via multiple paths and the association of privileges with individual named objects in a database. P-Pascal's bulk type simplified the storage and manipulation of security information, as well as the specification of an arbitrary number of passwords in a function call. The existence of sets permits statistical control, allows distinct privileges to be associated with specific operators and enables the protection of predicate-defined subsets. These privileges are difficult to provide for graph structures, since programs may have the right to manipulate an object but not be authorised to traverse any path to that data. Name equivalence ensures that protection is not lost on assignment; while the nesting of structures in P-Pascal complicates the handling of hidden components.

Chapter 8 described set representations and set evaluation strategies. Benefits of evaluating set expressions in a persistent programming language, rather than a relational database system, were identified. The advantages of building a bulk type into a language, rather than on top of the language, were outlined. The implementation of sets in PS-Algol and in Napier was discussed to show how the work undertaken in P-Pascal can be carried over into other persistent languages.

9.2 THE P-PASCAL CONTRIBUTION

This thesis has proposed a language for introducing orthogonal persistence into Pascal. A layered architecture for the abstract machine was presented and its implementation described. This showed how the CPOMS can support complex objects, and an interpreter with an architecture different from that for which it was designed. This persistence mechanism has been extended in three ways. Object clustering, metadata management and security enforcement are supported. These extensions can be adapted for similar persistence systems such as the Napier Stable Store [Brow89] in a relatively straight forward manner. A bulk data type in the form of a type complete set constructor has been implemented, as well as relational algebra operators. Implementation strategies used in relational database systems were seen to be applicable in a persistent language and additional optimisations for such an environment were identified. We propose that such high-level data manipulation be part of a persistent language, rather than being engineered on top of a language by the application programmers, in the interests of efficiency and ease of use. The techniques employed for P-Pascal sets can be applied to persistent languages with different data types and object representations.

9.3 FUTURE RESEARCH

There are several areas which require further work. A type system that includes sets (or relations) and first class pointers needs investigation from several viewpoints, such as programming methodology and database design. Research into extended relational systems has been in progress for some time [eg. Osbo, Ston83, Schm85], and the results apply equally to a persistent system supporting type complete sets. With regard to P-Pascal, the fourth layer of the architecture, relating to semantic constructs, has yet to be incorporated. Concurrency is clearly necessary, and first class functions and procedures are desirable. The language needs to be used in real-world applications; as Pascal is so well known, it is a good way of introducing programmers to persistent languages, and measuring the ease with which this concept can be assimilated. The methods used to introduce clustering, metadata management and security into the CPOMS could be implemented in its successor [Brow89]. The bulk data type developed for P-Pascal can be incorporated in a language with a richer type system.

An extensive study of persistent programming [Coop] has shown that this style of programming is an effective foundation for developing large, complex, long-lived

systems. Several aspects deserve further study however; in particular data evolution, distribution, hardware architectures and performance evaluation. Audit trails and high-level user interfaces are also needed. A formal model of persistent programming languages would increase our understanding of the issues and their interactions. Some work in this direction has been done by [Alba88, Ohor] and the author has begun work on a model based on subroutine automata [Berm91]. The amount of interest which persistent systems have attracted in recent years is encouraging, and it is clear that this research should continue to be pursued, leading to greater understanding of programming languages and long-term data management.

APPENDIX A P-PASCAL REPORT

1 DATA TYPES

A type is a simple type, or a structured type, or a pointer type.

1.1 SIMPLE DATA TYPES

1.1.1 PREDEFINED SIMPLE TYPES

The simple data types are the ordinal types and the predefined type REAL. Ordinal types define a set of values which can be mapped into some interval of the Integers, and as such their values are ordered. The types INTEGER, CHAR and BOOLEAN are predefined ordinal types.

Some examples:

```
VAR   Erroneous : BOOLEAN;
      Count  : INTEGER;
      Amt_Owed    : REAL;
      Reply  : CHAR;
```

1.1.2 USER DEFINED SIMPLE TYPES

User-defined types can be Simple or Structured. The simple ones are enumerated types and subrange types. The former introduces an ordered list of values represented by identifiers; the latter is specified by indicating the smallest and largest value in the subrange. Subranges can be defined on any of the ordinal types (i.e. pre- or user-defined).

Examples:

```
TYPE  Weekdays = (Monday, Tuesday, Wednesday, Thursday, Friday);
      Months = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
      Days = 1 .. 31;          { subrange of INTEGER }
      Group1 = 'A' .. 'M';     { subrange of CHAR }
      Summer_Months = Jun .. Aug;          { subrange of Months }
```

1.2 STRUCTURED TYPES

Instances of structured types can comprise several values. Such types are defined by specifying a structuring method and the types of the components. There are four structuring methods: array, record, set and file. All of these structures can optionally be PACKED to conserve space. A packed data type has the same rights as the equivalent unpacked type, with the exception that an actual VARIABLE parameter must not be a component of a packed structure. String types are packed arrays of CHAR. Recursive type definitions are not permitted, except in the case of the domain type of a pointer (see section 1.3).

1.2.1 ARRAY STRUCTURES

All array components are of the same type. This is known as their element type, and can be any P-Pascal type. Individual components or elements are selected by specifying index(es), also known as subscript(s). The type of an index must be ordinal. The number of elements in an array is fixed by giving the exact subrange of its index(es). An array can have any number of indexes (an n-index array is called n-dimensional). To reference a specific element, its index(es) are supplied after the name of the array, enclosed by square brackets, or by the symbols (. and .). Each index value must be assignment-compatible with the corresponding index type (see 1.2.4). The index expressions are evaluated from left to right.

Examples:

```
TYPE  arr1 = ARRAY [BOOLEAN] OF sometype;
      arr2 = ARRAY [Summer_Months] OF sometype;
      arr3 = ARRAY [1 .. 12, -3 .. 3, 'A' .. 'Z'] OF sometype;
```

The elements, of type "sometype", are:

```
      arr1 [FALSE]          arr1 [TRUE]

      arr2 [Jun]      arr2 [Jul]        arr2 [Aug]

      arr3 [1, -3, 'A']  arr3 [1, -3, 'B']   ...
      arr3 [12, 3, 'Z']
```

1.2.2 RECORD STRUCTURES

A record comprises a fixed number of components (or fields) distinguished by identifiers or field names. Fields can be of different types. The scope of a field is the record of which it is a component; hence field names need only be unique within this scope. Any P-Pascal type can be specified as the type of a field. A field order is associated with every record declaration; i.e. values in record instances appear in the order in which the fields are given in the record type definition. This ordering is utilised in the reading and writing of records, and in record aggregates. Each field declaration has the form

fieldname : type READKEY rkey WRITEKEY wkey

where rkey and wkey are strings, associating passwords with this component. The READKEY and WRITEKEY specifications are optional.

A record can include a variant part, which can be placed anywhere in the field ordering. It comprises an optional tag field and a list of variants. Each variant comprises any number (including zero) of fields of any types and is associated with a list of constants of the tag type, called case constants. A tag field must have a named ordinal type. No tag value may appear in more than one variant of a variant part, as the tag values serve to discriminate between the variants. If there is a tag field, the variant which is currently active is the one prefixed by the value in the tag field; otherwise the active variant is the one possessing the most recently accessed component.

A field is referenced by giving the name of a record, followed by a period and the name of the field. This is known as a field designator. ERROR_RECORD is a pre-defined record type used by persistence functions (see 8.5):

```
TYPE ERROR_RECORD = RECORD
                        error_num : INTEGER;
                        error_function : PACKED ARRAY [1..8] OF CHAR;
                        error_message : PACKED ARRAY [1..80] OF CHAR
     END;
```

Other examples:

```
TYPE person = RECORD
        name : surname_init_record;
        birthdate : date_record;
        height, weight : REAL;
        CASE status : 1..10 OF
                1,2,3,4 : ();
                5,6,7   : (hours, wage : REAL WRITEKEY 'y');
                8,9,10  : (salary : REAL READKEY 'x' WRITEKEY 'y';
                                start  : date_record;
                                dept   : ARRAY [1..20] OF CHAR )
        END
```

If variable R is of type person with a status of 5, 6 or 7, it would have the following fields:
R.name, R.birthdate, R.height, R.weight, R.status, R.hours, R.wage (in this order).

1.2.3 SET STRUCTURES

A set instance is an unordered collection of elements of the same type, called the base type. Unlike arrays, the size of a set is not fixed; furthermore indexes are not used to specify individual elements. The base type of a set can be any assignable P-Pascal type. A set never contains duplicate elements, i.e. two elements from the same set instance are always distinct. A set of ordinals is termed a Standard Set, as this is the only class of set permitted in Standard Pascal.

A set can have any number of keys specified (including zero): this identifies component(s) which uniquely identify its elements. If K is a key for a set S then no two elements of S can have the same value for K. A key can comprise any number of components of any assignable type (see 1.2.4) except set; however variant fields are not permitted, since key components must be defined for all elements of the set. In addition to specifying uniqueness constraints, keys are also used to construct indexes for fast access to sets. This can be utilised by the programmer to ensure that large sets are processed efficiently. Two keys for a set type can comprise identical components, given in a different order, and one can be a superkey of another.

For completeness sake, keys are specifiable for sets having a base type that is not a structure, although this is superfluous (elements only comprise one value).
Examples:

```
TYPE dayset = SET OF weekdays;
     mthset = SET OF Jun .. Nov;
     charset = SET OF CHAR;
     moneyset = SET OF REAL;
     payments = SET OF moneyset;
     class = SET OF stu_record KEY stu_name,stu_init;
                                   stu_number
                           END; (* keys *)
     staff = SET OF employee_record; (* no keys *)
     faculty = SET OF dept_array KEY [1] END;
```

1.2.4 FILE STRUCTURES

A file is a sequence of components of the same type. It has an associated mode which indicates whether the file is being generated or inspected. In either mode, file components are processed sequentially. P-Pascal files are not first class objects and cannot be placed on a database. Their components can be any assignable type. An assignable type is any type except a file or a structure with a component that is not assignable. The number of components in a file is not fixed at compile time. The predefined type TEXT is a file comprising a sequence of lines (where a line is a sequence of characters terminated by an end-of-line).

Only the current component of a file can be referenced in a program. This is done by referencing the buffer variable of that file, denoted filename^ or filename@.
Examples:

```
TYPE  file1 = FILE OF INTEGER;
      file2 = FILE OF student_record;
      file3 = TEXT;
```

1.3 POINTER TYPES

When dynamic data space is allocated on the P-Pascal heap, a reference to this data is returned. This is placed in a pointer variable, which is then used to refer to this object as

it has no associated name. A pointer value may only point to an object of one specific type (say T): the pointer is bound to type T. This type T can be any P-Pascal type. The pointer literal NIL is compatible with all pointer types. Every other value of type pointer must be obtained through the standard procedure New, which obtains an area of heap storage for the target object (called the identified object) and stores its address (called the identifying value) in the pointer variable. There are two pre-defined pointer types. Type DBPTR is the type of a pointer that points to a database. The type of the target object to which it points is hidden from programs. The other pre-defined pointer type is
TYPE ERROR_POINTER = ^ERROR_RECORD.

A pointer is dereferenced (to denote the target object) by following the pointer name with an up-arrow (an alternative notation for ^ is @).
Examples:

```
intptr  = ^INTEGER;
stu_ptr = ^student_record;
realptr = ^dept_array;
setptr  = ^moneyset;
ptrptr  = ^setptr;
```

1.4 TYPE COMPATIBILITY

Two types are compatible if any of the following holds:

(1) they are the same type (i.e. the same type name)
(2) both are string types with the same number of components
(3) one is a subrange of the other, or both are subranges of the same type
(4) both are Standard sets, their base types are compatible and either both are packed or neither is
(5) either is a derived type and they are structurally equivalent.

A derived type is created by projection, join, product or aggregation (see 3.2). Two records are structurally equivalent if they comprise the same number of fields, and the field types are compatible, compared pairwise in order. Two sets (or arrays) are structurally equivalent if their base types are compatible.

A value of type T2 can be assigned to a variable of type T1 (i.e. T2 is assignment-compatible with T1) if one of the following holds:

(1) T1 and T2 are the same assignable type
(2) T1 and T2 are compatible string types
(3) T1 is Real and T2 is Integer.
(4) T1 and T2 are compatible Standrad sets or compatible ordinal types and the value is a member of the set of values determined by T1
(5) T2 is a derived type which is compatible with T1.

2 OPERATORS

2.1 ARITHMETIC OPERATORS

Addition, subtraction, negation, multiplication, divison and remainder-on-division operators are defined for integers; denoted + - * DIV and MOD respectively. DIV rounds the quotient down to the nearest integer. Addition, subtraction, negation, multiplication and divison operators are defined for REALs; denoted + - * and / respectively. The INTEGER (REAL) operators are applicable to any value derived from type INTEGER (REAL), eg. some subrange thereof.

2.2 RELATIONAL OPERATORS

Equality (=) and Inequality (<>) is defined for values of any assignable type. Greater Than (>), Less Than (<), Less Than or Equal To (<=) and Greater Than or Equal To (>=) are defined for simple and string types. Set inclusion (denoted >= and <=), proper set inclusion (denoted > and <) and membership test (IN) are defined for sets of any type. The result type of a relational operator is always BOOLEAN.

Equality: Consider X = Y where X and Y are of type T. If T is a simple type, X = Y is TRUE if they represent the exact same number (or CHAR, or named value). If T is an array type, X = Y is true if X[index(es)] = Y[index(es)] for all values of the index(es). If T is a (variant) record type, X = Y is true if X.field = Y.field for all fields of T. If T is a set type, X = Y is true if every element of X is an element of Y and every element of Y is an element of X. If T is a pointer type, X = Y is true if X and Y point to the exact same object. Thus X^ = Y^ does not imply X = Y. Inequality is the converse of equality.

2.3 BOOLEAN OPERATORS

These are AND, OR and NOT. Their operands must be Boolean and the result is also a Boolean. If an operand is a Boolean expression, it must be enclosed by parentheses.

2.4 RECORD OPERATORS

A projection operator is defined on records, which defines a new record, possibly of a different type. The result can have any number of fields of any types and is specified field by field. This field specification must be enclosed in brackets after the projection operator ->. Structural equivalence is used to determine compatibility of a projection result with other record types. Within the brackets following a project operator, the fields can be denoted by their field name alone, similar to the body of a WITH statement (see WITH in 4.2.4). Alternatively they can be denoted by the full field designator (see 1.2.2).

Example:
sprec -> (sno, discount + 0.15, 'LA', datevar)
where sno and discount are fields of sprec
datevar is another program variable.
Or
sprec -> (sprec.sno,sprec.discount + 0.15,'LA',datevar)
which is equivalent to the first example.

2.5 SET OPERATORS

Operators which take operands of type set are union (denoted +), intersection (denoted *), set difference (denoted -), cartesian product (or "times", denoted #), ordering (denoted ORDER), subset selection (denoted by a condition enclosed in square brackets following the set expression) and desetting (denoted as for selection, but with [|...|] instead of square brackets). The operands of intersection, union and difference must be of compatible types.

2.5.1 CURSORS

To reference elements within a selection, desetting or projection, the set name is used directly as a cursor; and even this identifier can be omitted in many cases. The scope of a set identifier extends over unary set operators applied to it. A set expression accesses and establishes a reference to each set therein, and then executes these unary operators; the reference exists during the execution of these operators. Within this extended scope, the set variable identifiers are used to reference the current element (in database terms, they act as a cursor). If the current element is a record, its field identifiers can occur in a field designator without respecifying the record name, to denote the appropriate field of the referenced element.

The cardinality of the set is established prior to the execution of the selection, desetting or projection. Eg. in:

S[F(num) < 100]

the elements in S are fixed before the selection is applied, their values may subsequently be changed by function F, but F cannot cause new elements to be inspected or the cardinality of the cursor to be decreased. Any changes made to set S by function F are reflected in the value of S after the selection has terminated.

If A is a variable and A is also a fieldname of S elements, then the variable A and the set S are inaccessible in unary operators applied to S. Aliases are used to permit references to variables like A or to S itself, and are also used in expressions where the same set appears more than once. An alias is defined by following a set expression with "AS aliasname(aliaslist)". Either aliasname or aliaslist may be omitted; the latter is a list of identifiers separated by commas. This associates the elements of the expression with "aliasname" instead of the set variable name and/or associates fieldnames with data of derived types by means of an aliaslist.

Examples:

(X + Y) [cost < 9].

((X + Y) AS XY) [XY[1] < 9]

X->(wage+bonus,0) AS (pay,increment) JOIN Y

2.5.2 ORDERING

The ordering operator ORDER can be applied to any set expression, unless the base type is a set. This takes one or more ASC or DESC specifications giving the ascending (or descending) order in which elements are to be sorted. If the set has a base type that is a simple or set type, the ASC/DESC appears alone after the ORDER operator, eg

NUMSET ORDER DESC.

If several of these operators follow a set with a simple base type, the effect is the same as if the last one appeared alone.

If the set has a base type that is a (variant) record or array, ASC/DESC is followed by a single component specification, and any number of these specifications are possible. This enables the set to be sorted on primary key (given first), secondary key (given next), etc. Examples:

EMPSET ORDER ASC surname ASC initials

ARRSET ORDER DESC ARRSET[1,1].size

A field that is part of a variant may be used to specify the ASC/DESC ordering desired, but will cause the statement to fail if an element is encountered for which that specific variant is not active.

2.5.3 TIMES

The operands of Times (Cartesian Product) can be any two, possibly different, set types. The type of the result is always a set. The base type of elements produced by an n-way Times (i.e. S_1 # S_2 # ... # S_N) is an N-field record, where the j'th field is of type Tj, where Tj is the base type of Sj). The name of the i'th operand is taken as the i'th fieldname - aliases are used if fields require unique names for subsequent operator application. That is, if elements of S and R have a field name in common, it is not necessary to disambiguate the names in eg S # S, but will be required for eg. (S # S AS OTHER) [S.field < 10].

2.5.4 SELECTION AND DESETTING

The Select operator specifies a condition which elements of the set must satisfy if they are to appear in the result. The referencing of elements within the predicate is explained under CURSORS (see 2.5.1). The desetting operator is identical to select except that it

returns a single element satisfying the given predicate and its result is the same as the base type of the set. Desetting returns the first element encountered that satisfies the condition. The difference in syntax is that desetting uses [| ... |] instead of square brackets.

Examples:

([1,2,3,4,5] AS S) [S < 4] is [1,2,3]

(([1,2,3] - [3,4]) AS S) [S >= 2] is [2]

[1,2,3] # [4,5] is [[<1,4>], [<1,5>], ..., [<3,5>]]

([1,2,3,4,5] AS S) [| S < 3 |] is either 1 or 2 (integer)

(EmpSet # DeptSet) [(EmpSet.dno = DeptSet.dno) AND (wage > 90)]

In the latter example, it is not necessary to denote wage by EmpSet.wage unless DeptSet elements also have a wage field.

2.5.1 OPERATORS FOR SETS OF RECORDS

Natural Join (denoted JOIN) and Project (denoted ->) can be applied to operands that are sets of records. When Project is applied to such a set, any duplicates will be removed, so the cardinality of the original set may decrease. In all other respects, projecting over a set is identical to projecting over individual records (see 2.4 above).

JOIN can be applied to operands comprising sets of records. This operation is equivalent to Times, with the following differences:

(i) element combinations only appear in the result if they have the same value for all shared fields (where a shared field is a field having the same name in both operands' base types);

(ii) in an N-way join, the base type of the result set is a record comprising all the fields of operand 1, followed by all the non-shared fields of operand 2, ..., followed by all the non-shared fields of operand N. A non-shared field is any field that is not shared by an earlier operand.

If the operand elements associate the same field name with incompatible types, the JOIN result will be empty.

From (ii) we see that the result of X # Y and X JOIN Y are not the same, even if X and Y have no shared fields. This is because X # Y elements are records consisting of two record fields. If the result of a Cartesian product of two record sets must comprise

elements that are <fields of operand 1,fields of operand 2>, projection is used to restructure result elements.

Examples:

RecSet -> (field1, field2 DIV 2, field3, field4)

this updates every element, halving the field2 value; it may require duplicate elimination - eg. if originally there were only two elements

[< 1,3,5,7 >] and [< 1,2,5,7 >]

the result will have one element, viz.

[< 1,1,5,7 >].

(EmpSet JOIN DeptSet) [wage > 90]

this gives all data on high-earning employees, including their department details.

2.5.2 OPERATIONS FOR ARRAYS OF RECORDS

Projection may be applied to arrays of records. The effect is identical to projection on a set of records, except that duplicates are not discarded.

3 LITERALS AND AGGREGATES

3.1 SIMPLE LITERALS

The usual decimal notation is used for numbers, i.e. for INTEGERs and REALs. Exponential notation can be used by preceding the scale factor by the letter E.
Examples 100.23 1.0023E+2 -5E-4 etc.

Characters are enclosed by single quotes, as are strings.
Examples '?' 'String' 'Adam"s' etc. The two boolean literals are FALSE and TRUE. An enumerated type lists the values (identifiers) which are the literals of that type. The literals of a subrange type are denoted in the same way as literals of the type on which it is defined.

3.2 AGGREGATES

3.2.1 ARRAY AGGREGATES

An array aggregate is denoted by a list of elements enclosed by [. and .]. The first item in the list is placed in the first element of the array, etc. Row major order is used.
Examples:
[. 12, 34, 42, 5, 0 .]
[. 'A', 'E', 'I', 'O', 'U' .]
[. 'FIRST', 'SECOND', 'THIRD' .]

3.2.2 RECORD AGGREGATES

A record aggregate is denoted by a list of values enclosed by [< and >]. The first item in the list is placed in the first field of the record, etc.
Examples:
[<12,2,90>]
[< 'A', 1.32E8, [. 1,2,0,0,1 .], 'SMITH', [<12,2,90>] >]

3.2.3 SET AGGREGATES

A set aggregate is denoted by a list of elements enclosed in square brackets. The empty set is denoted by "[]".
Examples:

['a','c','e']	[[<1,2,89>], [<1,3,90>]]
[12.3, 4.3, 5.21]	[[. 1,3,4 .], [. 4,1,2 .], [. 1,2,2 .]]

3.2.4 FILE AGGREGATES

File aggregates are not supported.

3.3 POINTER LITERALS

There is only one pointer literal, NIL, which denotes the "empty" pointer, i.e. it does not point to any valid heap address. The NIL literal is not typed.

4 STATEMENTS

4.1 SIMPLE STATEMENTS

A simple statement is one which does not contain another statement. There are four such statements in P-Pascal:
assignments, procedure calls, goto statements and empty statements.

4.1.1 THE EMPTY STATEMENT

The empty statement contains no symbols and denotes no action.

4.1.2 THE ASSIGNMENT STATEMENT

This has the form

identifier := expression

where identifier is the name of a variable or function. The type of the identifier and the expression must be assignment compatible.

4.1.3 PROCEDURE CALLS

These serve to execute the named procedure with the given parameters as actual arguments (also called actual parameters), if any. Arguments must be given in the same order as in the corresponding procedure declaration, and no parameter may be omitted.

A value or variable parameter can be of any P-Pascal type. Unlike value parameters, variable parameters must supply variable names as actual arguments. The programmer should bear in mind that value parameters are assigned to separate, local objects inside the called procedure; this should be carefully considered before large arrays or sets are used as value parameters. A variable parameter can be a dereferenced object; it can be a structured object; it can also be a single component of a structured object. Neither a tagfield nor a packed array element may act as an actual VAR argument.

A conformant array parameter is one defined by means of an index specification that has identifiers in place of constants. If a formal value parameter is not a conformant array parameter, the value of the actual parameter must be assignment-compatible with the

type of the formal parameter. If the formal value parameter is a conformant array parameter, the type of the actual parameter must not be a conformant type. If a formal variable parameter is not a conformant array parameter, then the actual parameter and the formal parameter must possess the same type.

An array type T (with a single index type) is said to be conformable [J&W] with a conformant array schema S (with a single index type specification) if all of the following are true (let I represent the ordinal type identifier of the index type specification of S):

- the index type of T is compatible with the type denoted by I
- every value of the index type of T is a member of the set of values of the type denoted by I
- if S does not contain a conformant array schema, then the component type of T is the same as the type denoted by the type identifier in S; otherwise, the component type of T is conformable with the component schema of S.
- T is packed if and only if S is a packed conformant array schema.

If a variable parameter is an array element, its index is evaluated when the procedure is called, i.e. before the procedure body is executed. Similarly, if it is a set element, the target element is determined (i.e. the desetting is performed) when the procedure is called. An actual function parameter must have a result type identical to that specified by the formal parameter. An actual procedure (or function) parameter must be such that its parameters are congruent with the parameters in the corresponding formal parameter.

Two formal parameter lists are congruent if they have the same number of parameter sections, and each satisfies one of the following:

(1) both specify value parameters and comprise the same number of identifiers, which are either of the same type or are equivalent conformant array schemas;

(2) both specify variable parameters and comprise the same number of identifiers, which are either of the same type or are equivalent conformant array schemas;

(3) both are procedural parameter specifications with congruent parameter lists;

(4) both are functional parameter specifications with congruent parameter lists and the same result type.

4.1.4 THE GOTO STATEMENT

This statement directs execution to some labelled statement within the same procedure. It is included only for upward compatibility with Standard Pascal.

4.2 STRUCTURED STATEMENTS

4.2.1 COMPOUND STATEMENTS

A compound statement is a collection of P-Pascal statements enclosed by BEGIN and END. This makes them a single statement, which can then be used within structured statements.

4.2.2 CONDITIONAL STATEMENTS

(i) IF <expression> THEN <statement1> ELSE <statement2>
and
(ii) IF <expression> THEN <statement1>

evaluate the given Boolean expression and execute statement1 if it is True. Otherwise, in (i) statement2 is executed; in (ii) execution proceeds with the statement following the IF. When IF statements are nested, an ELSE is interpreted as being part of the last IF statement which does not already have an ELSE part.

CASE <expression> OF
<list> : <statement> ;
- - -
<list> : <statement>
END

causes the result of expression to be located in one of the lists, and the corresponding statement executed. Each list comprises one or more literals, having the same type as the expression result. No literal may appear more than once in these lists. At runtime, the program fails if the expression value does not appear in any list. The expression and list values must be of the same ordinal type.

4.2.3 REPETITIVE STATEMENTS

(1) WHILE <expression> DO <statement>
(2) REPEAT <statement> UNTIL <expression>
(3) FOR <control-variable> := <expression1>
TO <expression2> DO <statement>
(4) FOR <control-variable> := <expression1>
DOWNTO <expression2> DO <statement>
(5) FOREACH <expression> DO <statement>

In a WHILE and REPEAT loop, the expression must return a Boolean result. The former performs <statement> repeatedly as long as this evaluates to True; the latter executes <statement> repeatedly until this is False. A REPEAT always causes <statement> to be performed at least once (before the <expression> is evaluated); a WHILE loop tests <expression> first and thus may not perform <statement> at all.

In a FOR loop, the <statement> may not be executed at all, as the effect is to assign to <control-variable> the value of <expression-1>, then test it against <expression2> before performing <statement> and assigning it a new value. In (3) the new value is the successor of the current value, in (4) it is the predecessor. The loop terminates when the <control-variable> exceeds <expression2> in the case of (3), or is less than <expression2> in the case of (4). In any FOR loop, the <control-variable>, <expression1> and <expression2> must be of the same ordinal type. After exiting the FOR loop, the value of the <control-variable> is undefined. A statement S threatens a variable V if S assigns to V, or S contains V as an actual variable parameter, or S is a procedure statement that activates Read or Readln with V as parameter or S is a FOR statement and V is its control variable. Niether the loop body nor any statement in a procedure or function local to the block may threaten the <control-variable>, which must be a variable local to the block in which the FOR statement appears. This ensures that the repeated statement cannot alter the value of the control variable [Jens].

In a FOREACH statement, the <expression> must yield a result of type set or array. The <statement> will be executed once for each element encountered in this expression. Within <statement>, the current element of <expression> is referenced using cursors, as in select, desetting or project. When the <expression> is a variable it may appear as

L_value within the loop (i.e. it may be assigned a value). This is also possible if it is a set variable that has unary set operators applied to it. Example:

FOREACH S [status = x] DO S.value := S.value * 2;

4.2.4 THE WITH STATEMENT

WITH <record variable list> DO <statement>

The effect of this statement is to permit field names of the listed records to be used directly within <statement>, without preceding this with the record name.

5 PROCEDURE DECLARATIONS

A Procedure declaration comprises a heading and a block. As in a program, a procedure block can include LABEL, CONST, TYPE, VAR, PROCEDURE and FUNCTION declarations in addition to the statement defining its action(s). In the current version of P-Pascal, a PROCEDURE is not a first-class object: it may not be part of a structured type, nor be made persistent.

The procedure heading gives the procedure name and defines the formal parameters (if any). Each formal parameter is associated with a named type, which can be any P-Pascal type. A parameter is one of the following kinds: value, VAR, function or procedure. Where a parameter is a function or procedure, its parameters are also specified, as can be seen from the following examples:

```
PROCEDURE sometask (VAR amt:REAL; PROCEDURE prc1(X : real));
PROCEDURE action (FUNCTION f(A:real):real; a1,a2: bigarray);
```

A procedure can alternatively be declared using the FORWARD directive. This comprises the procedure heading followed by the directive FORWARD. The body is declared at a subsequent point in the program, preceded by the word PROCEDURE and the procedure name (parameterlist omitted).

6 STANDARD PROCEDURES

6.1 FILE HANDLING PROCEDURES

The PUT, GET, RESET and REWRITE procedures of Standard Pascal are retained by P-Pascal.

6.2 DYNAMIC ALLOCATION PROCEDURES

The DISPOSE and NEW procedures of Standard Pascal are retained by P-Pascal, for the (de-) allocation of heap space. Both these procedures have a single pointer parameter, which can have any type as domain. Dispose is retained for upward compatibility with Standard Pascal and has not effect; heap garbage collection is under system control.

6.3 DATA TRANSFER PROCEDURES

The PACK and UNPACK procedures of Standard Pascal are retained by P-Pascal.

7 FUNCTION DECLARATIONS

A Function declaration comprises a heading and a block. As in a program, a function block can include LABEL, CONST, TYPE, VAR, PROCEDURE and FUNCTION declarations, in addition to the statement defining its action(s). In the current version of P-Pascal, a FUNCTION is not a first-class object: it may not be part of a structured type, nor be made persistent.

The function heading gives the function name and defines the formal parameters (if any); it also specifies the (named) type of the result. The value returned by a function can be any assignable P-Pascal type. The rules for parameter specification are the same as for Procedure declarations.
Examples:

```
FUNCTION sometask (VAR amt:REAL; PROCEDURE prc1(A:real) )
                  : some_set_type;
FUNCTION action (FUNCTION f(x:real):real; a1,a2: bigarray)     : bigarray;
```

A function can alternatively be declared using the FORWARD directive. This comprises the function heading followed by the directive FORWARD. The body is declared at a subsequent point in the program, preceded by the word FUNCTION and the function name. In other words, the result type and any parameterlist are then omitted.

8 STANDARD FUNCTIONS

Functions to query metadata (persistent or otherwise) are given in Appendix C. The remaining functions are given below.

8.1 ARITHMETIC FUNCTIONS

The ABS, SQR, SIN, COS, EXP, LN, SQRT and ARCTAN functions of Standard Pascal are retained by P-Pascal.

8.2 BOOLEAN FUNCTIONS

The ODD, EOF and EOLN functions of Standard Pascal are retained by P-Pascal.

8.3 TRANSFER FUNCTIONS

The TRUNC, ROUND, ORD and CHR functions of Standard Pascal are retained by P-Pascal.

8.4 OTHER STANDARD FUNCTIONS

The SUCC and PRED functions of Standard Pascal are retained by P-Pascal.

8.5 PERSISTENCE FUNCTIONS

The six functions dealing with persistence are COMMIT, CREATEDB, OPENDB, SAVE, DROP and LOCATE.

COMMIT has no parameters. Its effect is to update all open databases to reflect changes made by the program. This is an atomic action, in that either all changes will be made, or (in the case of failure) none.

SAVE makes a data object persistent. Such an object is termed a "primary object" or "database primary", as it is explicitly placed on a database and associated with a unique name. The transitive closure of all pointer references in a primary is used to identify further persistent objects (referential integrity on the persistent store is thus guaranteed). Save has six parameters (in this order): a pointer to the database where the object is to be stored, the name of the object, a pointer to the object and three optional passwords to protect the data. These are (in this order:) a read password for the object, a write password and component passwords. The object name, read password and write password are strings. The database pointer is of standard type DBPTR. The pointer argument can be any pointer type. The component passwords are given as a set of strings. Each element has the form 'fieldname:mode:password', where mode is R (for read) or W (for write).

LOCATE retrieves a primary object from a database. It has six parameters (in this order): a pointer to the database where the object resides, the name of the primary, a pointer indicating where the object is to be placed on the heap, and three optional passwords to authorise the access - first a read password, then a write password and finally component passwords. The arguments are analogous to their SAVE counterparts.

DROP removes a primary object name from a database. If no other persistent data references this object, it is deleted at the next persistent store garbage collection. It has five parameters (in this order): a pointer to the database where the object resides, the name of the object and three optional passwords to authorise the removal - first a read password, then a write password and finally component passwords. The arguments are analogous to their SAVE counterparts.

CREATEDB creates a new database in the persistent store. Its parameters (in order) are: a DBPTR, the database name (a string), the user-id of the database owner (a string), two integer parameters, a metadata-flag, and a read- and a write-password for the database (both strings). The last two parameters are optional. A pointer to the new database is returned in the DBPTR argument. The first integer is the placement factor, which must be between zero and 100. Once this percentage of an indexfile block has been filled, the remainder will be reserved for objects related to its occupants. It is zero if no clustering is required at all, and is set to 100 if objects are to be clustered by type but not by family. The other integer gives the loadfactor for sets, which specifies the proportion of each disk

block allocated to sets which must be reserved for future elements. The flag parameter is a Boolean, indicating whether or not cross-referencing information should be maintained for the database.

OPENDB opens an existing database in the persistent store. It has a DBPTR parameter followed by a database name, mode, read- and write-password for the database. The last two parameters are optional. All parameters are strings except for the mode, which is a character. This is 'R' if the database is being opened in read mode, otherwise it is 'W'. A pointer to the opened database is returned in the DBPTR argument, and is used to Locate or Save objects on that database. No data may be retrieved from or saved on a database prior to its OPENing.

All persistent functions return an ERROR_POINTER, which is a pointer to an ERROR_RECORD, or NIL on successful termination.

9 INPUT/OUTPUT

9.1 READ(LN)

The READ(LN) procedures of Standard Pascal are retained by P-Pascal; but extended to admit record, set and array arguments. Values are read into variables according to the type of the variable:

INTEGER, REAL or subrange thereof: the next number in the file will be located and placed in the target variable;

CHAR or subrange thereof: the next character in the file will be placed in the target variable (any character is valid here);

ARRAY of N elements: the next N values are read from the input file and placed in each element in turn (row major order);

(VARIANT) RECORD: values from the input file are assigned to each field in turn, in the order in which these fields are defined - pointer and enumerated fields are omitted, since their values cannot be input;

SETS: a sequence of values enclosed by square brackets is read from the input file and inserted into the set (duplicates and elements that duplicate keys are ignored). If "]" is required to be in the set, a double bracket i.e. "]]" is used.

9.2 WRITE(LN)

The WRITE(LN) procedures of Standard Pascal are retained by P-Pascal; but extended to admit array, set and record arguments. Values are written out according to type:

INTEGER, REAL or subrange thereof: as for Standard Pascal;

CHAR or subrange thereof: as for Standard Pascal;

ARRAY of N elements: the N element values are written to the output file in turn (row major order);

(VARIANT) RECORD: each field's value is written to the output file in turn, in the order in which these fields are defined in the type declaration - pointer and enumerated fields are omitted, since their values cannot be printed;

SETS: the elements are written to the output file as a sequence of values.

In addition to the formatting supported by Standard Pascal (i.e. :width, or :width :decimaldigits for REALS), bracketed format specifications can be given for structured data.

ARRAYS: If the array expression is followed by a colon and a formatting expression in brackets, the bracketed formatting is applied repeatedly until the elements of the array have all been printed.

(VARIANT) RECORDS: If the record expression is followed by a colon and a formatting expression in brackets, the bracketed formatting is applied repeatedly until all the fields of the record have been printed - no formatting must be given for pointer and enumerated fields, as these are ignored by WRITE(LN);

SETS: If the set expression is followed by a colon and a formatting expression in brackets, the bracketed formatting is applied repeatedly until the elements of the set have all been printed.

If the formatting specified for a variant record requires that real formatting be applied to a non-numeric value, an error results.

EXAMPLE:

```
WRITELN (intarr : (:5, :10),  rec_o_nums : (:5:2, :9:0, :5),  intset : (:10) );
```

9.3 OTHER I/O PROCEDURES

The PAGE procedure of Standard Pascal is retained by P-Pascal.

10 LABELS

An optional label declaration part can be associated with a program, procedure or function. It specifies all labels which mark statements in the statement part of the block. Its syntax is

LABEL labellist

where labellist is a sequence of labels separated by commas. A label is an unsigned integer of at most 4 digits. A statement is labelled by preceding it with a label followed by a colon. The GOTO statement transfers control to a labelled statement. The scope of a label is the procedure/function/program in which is it defined; it is not possible to jump into a procedure or function.

11 CONSTANT DEFINITION

An optional constant definition part can occur in a program, procedure or function, between the label declaration part and the type definition part. This introduces an identifier as a synonym for a value. The syntax is

CONST constantlist

where the constantlist comprises constant declarations separated by semicolons. Each has the form

identifier = value

The value can be of any type; if it is structured, it is given as an aggregate (see 3.2).

12 TYPE DECLARATIONS

An optional type declaration section can occur in a program, procedure or function between the constant and variable declarations. The syntax is

TYPE typedefinitions

where typedefinitions is a list of type declarations separated by semicolons. Each associates a name with a type, and has the form

identifier = type READKEY rkey WRITEKEY wkey

where type is any valid Pascal type, as described in section 1, and rkey and wkey are strings. These are optional and specify a password for accessing and for changing/creating objects of the type respectively.

13 VAR DECLARATIONS

An optional VAR declaration part can occur in a program, procedure or function between the type and procedure/function definition sections. The syntax is

VAR declarationlist

where declarations are separated by semicolons and have the form

identifierlist : type

Any valid P-Pascal type definition or type name can be used. The identifierlist is a list of identifiers separated by commas. Every variable must be declared in a VAR declaration before it is used; the identifier/type association is valid within the block containing this declaration.

14 PROGRAMS

A program comprises a program heading and a block. The program heading contains the following: the word PROGRAM followed by an identifier giving the program name, followed by a list of identifiers for the external files used by the program. A block comprises optional LABEL, CONST, TYPE, VAR, PROCEDURE and FUNCTION declarations, followed by a statement giving the action(s) of the program. These components of a block must appear in exactly this order - the only flexibility is that PROCEDURE and FUNCTION declarations can be intermingled.

15 COMMENTS

A Comment is any text preceeded by the symbol { or (* and followed by the symbol } or *). Comments may not be nested, hence the symbols } and *) cannot be part of a comment.

APPENDIX B THE TYPE SYSTEM OF P-PASCAL

1 Introduction

The P-Pascal type system is characterised according to the framework presented in [DBPL2].

2 The Nature of the Type System

2.1 Does the language have a type system? What is the model of type?

The model of a type is a set of values.

2.2 What is the purpose of the type system?

Types are used for protection and information modelling. Protection is afforded by strong type-checking of transient and persistent data. Types can be used as a basis for clustering objects and for associating user-defined security privileges. Implementation details are apparent in the distinction between packed and unpacked structures: the former are represented more compactly but are less efficient to access. Other implementation details such as the existence of indexes and the internal representation of structures are transparent.

2.3 Is the language strongly typed? Is type checking static or dynamic?

Strong typing and eager type-checking is employed; this is performed statically except for the standard functions Locate and Read(ln) which obtain objects from the persistent store and from input/output files respectively. Deep type-checking occurs on a Locate.

2.4 How is type checking used in data definition and for operations?

Type declarations are checked for consistency and operations are strictly type-checked.

3 Expressiveness

3.1 What primitive types and type constructors are available in the language?

The primitives types are integer, boolean, char and real. The pointer, subrange and enumerated types of Pascal are retained, as are the type constructors array, record, variant record, file and set. A set can have any number of keys defined, in which case its elements will never contain duplicate values for any key. Type constructors can optionally be packed.

3.2 Are there restrictions on the combinations of type constructors or the form of recursion?

Recursive type declarations are not supported. The only restriction on type combinations is that the base type of a set must be an assignable type, as these are the first class types in the language. A type is assignable if it is not a file and all its components are assignable.

3.3 What kinds of polymorphism are supported by the type system?

Only ad hoc polymorphism is supported, in the form of overloaded assignment, arithmetic and relational operators. Certain standard functions are polymorphic: Locate, Save, Read(ln) and Write(ln). Several set operators are also polymorphic (membership test, relational expressions, selection, product, intersection, union, difference, desetting, ordering).

4 Types and values

4.1 What are the properties that are possessed by values of all types? Is there a concept of first class values? What are the properties that define first class values? Do all types have these properties?

Procedures and functions are not first class objects. All assignable types are first class objects; having the right to be declared, to be assigned to and to be assigned, to be passed as parameters and returned as function results and to persist. The principles of persistence independence and data type completeness are employed.

Values of any type can be constructed, from simple literals to array, record or set aggregates.

4.2 Are there values that can be typed that have special properties not shared by all values? What are they and how are they special?

No.

4.3 Does the type system allow the construction of objects?

All values are objects in that they have state, identity and operations.

4.4 Is every value an object, or do both objects and non-object values exist? Are objects first class values? What is the exact distinction (if any) between objects and first class values in the language? How do types of objects fit into the type system?

All values are objects. However, the language is not object-oriented, as eg. inheritance is not supported.

4.5 What is the semantics of the equality relation?

Equality is based on identical value, not on identity.

4.6 Is there a concept of null value? Are there different kinds of null values? How do they interact with the equality relation?

The literal NIL and the aggregate [] (the empty set) are the only values of type null.

4.7 Can a value have more than one type simultaneously? Can a value change its type? What are the restrictions on such mutations?

As in Standard Pascal, if T1 is a subrange type defined on type T2, then all values of type T1 are also of type T2.

5 Relationships among types

5.1 What is the nature of type equivalence?

Type equivalence is name equivalence, with two exceptions. When a Locate is executed, the name and structure of the corresponding type on the persistent store is checked against the program type. Derived record types are compatible if they have compatibale field types, compared pairwise in order.

5.2 Does the language offer abstract, "concrete" and "unnamed" types?

All type declarations define concrete types. There are no abstract types. Variables and structure components can be of unnamed type.

5.3 How are abstract types related to object types?

There are no abstract types.

5.4 Does the language offer a module mechanism and, if so, what are its features? Are modules related to abstract data types? Are modules realised through a type constructor or are they a concept unrelated to the type system?

The present implementation has no module mechanism. The persistent store is structured as a number of inter-linked databases.

5.5 Is there a concept of abstract type in which the term "abstract" refers to the stipulation that the type is not meant to have instances, but only to allow for the more convenient definition of other types?

No.

6 Types and subtypes

There is no notion of subtyping, except that a subrange of type T is a subtype of T.

7 Classes and subclasses

There is no concept of classes.

8 Database issues

8.1 Is persistence orthogonal to the type system?

Yes.

8.2 What kinds of database schema evolution are supported?

There is currently no special support for schema evolution.

8.3 Are there issues of transaction processing, concurrency control, resilience, reliability or recovery that are addresses in the type system?

No.

9 Other issues

9.1 To what extent is type inference employed in the language?

P-Pascal inferences the type of expressions.

9.2 Is there any theory supporting the type system?

No.

9.3 Are there implementation factors that are essential to the understanding of the system?

No.

9.4 Are there issues that are not covered in the previous questions that are important to understanding the type system?

Not that we know of.

9.5 What is the status of the implementation of the system?

An initial version of P-Pascal has been implemented.

APPENDIX C METADATA ACCESS FUNCTIONS

The P-Pascal metadata access functions are listed here, of which only a representative sample have been implemented, to demonstrate their viability. The meaning of some functions are given in the list, the rest should be obvious from the function names and the conventions described below.

All functions use the following conventions:
scope indicates to what the function must be applied, and can be one of six possibilities:
"S" (persistent **S**tore as a whole), "D" (the single **D**atabase named in the db parameter), "U" (all **U**sable types, i.e. all types on open database as well as transient types), "O" (all **O**pen databases), "R" (all databases opened for **R**eading) or "W" (all databases opened for **W**riting);
db is the name of the database of interest - its contents are irrelevant if the scope is other than "D";
err is of type ERROR_POINTER, and is NIL if the function succeeds;
flag is either "R" if root objects are of interest or "T" if types are of interest;
name is the name of the root or type of interest;
field is the name of a field;
offset is the offset of a field from word zero of the record;
dimension is an integer indicating the first (most significant) subscript, or the second, etc.

Boolean Functions:

IS_PROTECTED(scope,flag,name,err) {does this name (of a root or a type, depending on flag) have (read or write) passwords associated with it} ,
IS_READ_PROTECTED(scope,flag,name,err) {as above, but only readkeys of interest},
IS_WRITE_PROTECTED(scope,flag,name,err),
IS_OWNER(scope,flag,name,userid,err) {is this user the owner of "name" ?},
IS_ELM_NAMED(scope,flag,arr_or_set_name,err) {is the element type of the set or array named ?},
IS_FIELD_NAMEDTYPE(scope,flag,name,field,err),
IS_FIELD_FIXED(scope,flag,name,field,err),
IS_FIELD_TAG(scope,flag,name,field,err),
IS_PACKED(scope,flag,name,err),

BOOLEAN_LOWER_DIM(scope,flag,name,dimension,err) {name an array with BOOLEAN index range},
BOOLEAN_UPPER_DIM(scope,flag,name,dimension,err) {name an array with BOOLEAN index range},

IS_EMPTY(scope,flag,name,err,db) {are there no occurrences of "name" ?},
IS_CLUSTERED(scope,flag,db,err,db),
IS_PROTECTED(scope,flag,db,err,db), IS_READ_PROTECTED(scope,flag,db,err,db),
IS_WRITE_PROTECTED(scope,flag,db,err,db),
EXISTS(scope,flag,name,err,db) {is this the name of a type (or root, depending on flag) in the persistent store ?}

String Functions:

KIND_OF(scope,flag,name,err,db) {returns "char", "integer", "real", "boolean", "pointer", "enumerated", "subrange", "set", "array", "record", "variant" or "file" },
OWNER_OF(scope,flag,name,err,db),
WRITE_PWD_OF(scope,flag,name,userid,ownerpwd,err,db) {returns write password, if userid and ownerp(ass)wd correctly identify user as the owner of "name" },
READ_PWD_OF(scope,flag,name,userid,ownerpwd,err,db),
FIELD_NAME_OF(scope,flag,name,offset,err,db),
FIELD_READ_PWD_OF(scope,flag,name,userid,ownerpwd,field,err,db)
FIELD_WRITE_PWD_OF(scope,flag,name,userid,ownerpwd,field,err,db),
ELM_KIND_OF(scope,flag,arr_or_set_name,err,db),
ELM_NAME_OF(scope,flag,arr_or_set_name,err,db),
DIM_KIND_OF(scope,flag,name, dimension,err,db),
FIELD_KIND_OF(scope,flag,name,field,err,db),
TAG_KIND_OF(scope,flag,name,field,err,db),
ENUM_VALUE(scope,flag,name,n,err,db) {the n'th value of the enumerated type},
ENUM_LOWER_DIM(scope,flag,name,dimension,err,db) {name of lower subscript, where index range is enumerated type},
ENUM_UPPER_DIM(scope,flag,name,dimension,err,db) {name of upper subscript, where index range is enumerated type},
ENUM_LOWER_BOUND(scope,flag,name,err,db) {subrange of enumerated type},
ENUM_UPPER_BOUND(scope,flag,name,err,db) {subrange of enumerated type}

Integer Functions:

NUM_DIMENSIONS(scope,flag,name,err,db) {name is an array root/type},
INT_LOWER_DIM(scope,flag,name,dimension,err,db),
INT_UPPER_DIM(scope,flag,name,dimension,err,db),
INT_LOWER_BOUND(scope,flag,name,err,db) {of a subrange type},
INT_UPPER_BOUND(scope,flag,name,err,db) {of a subrange type},
NUM_FIELDS(scope,flag,name,err,db),
NUM_FIXED_FIELDS(scope,flag,name,err,db),
NUM_VARIANT_FIELDS(scope,flag,name,err,db),
RECORD_SIZE(scope,flag,name,err,db) {in words},
FIXED_SIZE(scope,flag,name,err,db) {size in words of fixed fields},
MAX_VARIANT_SIZE(scope,flag,name,err,db) {maximum size in words of variant part},
NUM_PROTECTED_FIELDS(scope,flag,name,err,db),
NUM_READ_PROTECTED_FIELDS(scope,flag,name,err,db),
NUM_WRITE_PROTECTED_FIELDS(scope,flag,name,err,db),
NUM_FIELDS_OF(scope,flag,name,fieldtype,err,db) {no. of fields of name which are of type fieldtype},
NUM_FIXED_FIELDS_OF(scope,flag,name,fieldtype,err,db),
NUM_VARIANT_FIELDS_OF(scope,flag,name,fieldtype,err,db),
NUM_TAG_FIELDS_OF(scope,flag,name,fieldtype,err,db),
NUM_ENUM_VALUES(scope,flag,name,err,db) {name is an enumerated type},

NUM_INSTANCES(scope,flag,name,err,db) { counts occurrences of "name" on clustered databases only },
NUM_OF(scope,flag,err,db) {number of root objects (or types, depending on flag) },
NUM_PROTECTED(scope,flag,err,db),
NUM_READ_PROTECTED(scope,flag,err,db),
NUM_WRITE_PROTECTED(scope,flag,err,db),
NUM_OWNERS(scope,flag,err,db) {number of owners of root objects (or types, depending on flag) },
NUM_NAMES_OWNED(scope,flag,userid,err,db),
NUM_SETTYPES_OF(scope,flag,name,err,db) {this gives no. of types (or roots) on the persistent store that are "SET OF name"},

NUM_ARRAYTYPES_OF(scope,flag,name,err,db),
NUM_FIELDS_OF(scope,flag,name,err,db),
NUM_FIXED_FIELDS_OF(scope,flag,name,err,db),
NUM_VARIANT_FIELDS_OF(scope,flag,name,err,db),
NUM_RECORDS_WITH(scope,flag,name,err,db) {no. of record types in the store that include one or more fields of this type},
NUM_RECORDS_WITH(scope,flag,db,name,err,db)

Set Functions:

ALL_NAMES(scope,flag,err,db) { the names of all root objects (or types, depending on flag) },
PROTECTED_NAMES(scope,flag,err,db) { the names of all protected root objects (or types, depending on flag) },
READ_PROTECTED_NAMES(scope,flag,err,db),
WRITE_PROTECTED_NAMES(scope,flag,err,db),
OWNERS(scope,flag,err,db), NAMES_OWNED(scope,flag,userid,err,db),
SETS_OF(scope,flag,name,err,db) {this gives the types (or roots) on the store that are "SET OF name"},
ARRAYS_OF(scope,flag,name,err,db) {this gives the types (or roots) on the store that are "ARRAY [..] OF name"},
FIELDS_OF(scope,flag,name,err,db) {this gives the fields of roots (or types) of the given "name"},
FIXED_FIELDS_OF(scope,flag,name,err,db),
VARIANT_FIELDS_OF(scope,flag,name,err,db),
RECORDS_WITH(scope,flag,name,err,db) {record types in the persistent store that include one or more fields of type "name"},

DIMENSIONS(scope,flag,name,err,db),
INT_LOWER_DIMS(scope,flag,name,dimension,err,db),
INT_UPPER_DIMS(scope,flag,name,dimension,err,db),
FIELDS_OF(scope,flag,name,err,db),
FIXED_FIELDS(scope,flag,name.err,db), VARIANT_FIELDS(scope,flag,name,err,db),
TAG_FIELDS(scope,flag,name,err,db), INTEGER_FIELDS(scope,flag,name,err,db),
REAL_FIELDS(scope,flag,name,err,db), CHAR_FIELDS(scope,flag,name,err,db),
BOOLEAN_FIELDS(scope,flag,name,err,db),

POINTER_FIELDS(scope,flag,name,err,db), ARRAY_FIELDS(scope,flag,name,err,db),
RECORD_FIELDS(scope,flag,name,err,db),
VARIANT_RECORD_FIELDS(scope,flag,name,err,db),
SET_FIELDS(scope,flag,name,err,db),
INT_VARIANTS(scope,flag,name,err,db) {variants of "name" records with integer tag},
CHAR_VARIANTS(scope,flag,name,err,db) {variant of "name" records with char tag},
BOOL_VARIANTS(scope,flag,name,err,db) {variant of "name" records with boolean tag},
VALUES_OF(scope,flag,name,err,db) {"name" is an enumerated type},
PROTECTED_FIELDS(scope,flag,name,err,db),
READ_PROTECTED_FIELDS(scope,flag,name,err,db),
WRITE_PROTECTED_FIELDS(scope,flag,name,err,db)

Char Functions:

CHAR_LOWER_DIM(scope,flag,name,dimension,err,db),
CHAR_UPPER_DIM(scope,flag,name,dimension,err,db),
CHAR_LOWER_BOUND(scope,flag,name,err,db) {of a subrange type},
CHAR_UPPER_BOUND(scope,flag,name,err,db) {of a subrange type}

REFERENCES

[Abit] Abiteboul, S. and Hull, R., "IFO: A formal semantic database model", ACM Trans. on Database Systems 12(4), pp. 525-565, 1987.

[Abri] Abrial, J.R., "Data Semantics" in Data Base Management, North-Holland, pp. 1 - 59, 1974.

[Agra] Agrawal, R. and Genahni, N.H., "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language)++", Proc. 2nd Int. Workshop on Database Programming Languages, pp. 25-40, 1989.

[Aho] Aho, A.V., Sethi, R. and Ullman, J., "Compilers: Principles, Techniques and Tools", Addison-Wesley, Reading MA, 1986.

[Alba85a] Albano, A., Cardelli, L. and Orsini, R., "Galileo: A Strongly-Typed, Interactive Conceptual Language", ACM TODS 10(2), June 1985.

[Alba85b] Albano, A., Ghelli, G. and Orsini, R., "The Implementation of Galileo's Values Persistence", Proc. 1st Appin Workshop on Persistence and Data Types, Scotland 1985, PPRR-16, pp. 197-208.

[Alba85c] Albano, A., Ghelli, G., Occhiuto, M.E. and Orsini, R., "Galileo Reference Manual Version 2.0", Dipartimento di Informatica, Universita degli Studi di Pisa, 1985.

[Alba88] Albano, A., Giannoti, F., Orsini, R. and Pedereschi, D., "The Type System of Galileo", in "Persistence and Data Types", Atkinson, M., Buneman, O.P. and Morrison, R. (editors), Springer-Verlag, pp. 101-119, 1988.

[Alba89a] Albano, A., Dearle, A., Ghelli, G., Marlin, C., Morrison, R., Orsini, R. and Stemple, D., "A Framework for Comparing Type Systems for Database Programming Languages", Proc. 2nd International Workshop on Database Programming Languages, Oregon, 203-212, June 1989.

[Alba89b] Albano, A., Ghelli, G. and Orsini, R., "Types for Databases: The Galileo Experience", Proc. 2nd International Workshop on Database Programming Languages, Oregon, 196-206, June 1989.

[Ambl] Amble, T., Bratbergsengen, K. and Risnes, O., "Astral: A Structured and Unified Approach to Database Design and Manipulation", Proc. of Database Architecture Conf., Venice, Italy, 1979.

[Atki81] Atkinson, M.P., Chisholm, K.J. and Cockshott, W.P., "PS-Algol: an Algol with a Persistent Heap", ACM SIGPLAN Notices 17(7), July 1981.

[Atki83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R., "An Approach to Persistent Programming", Computer Journal 26(4), 1983.

[Atki84] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R., "Progress with Persistent Programming", in "Database - Role and Structure", CUP, 1984.

[Atki85] Atkinson, M.P. and Buneman, O.P., "Database Programming Language Design", University of Glasgow and University of St Andrews PPRR 17, 1985.

[Atki86] Atkinson, M.P. and Morrison, R., "Integrated Persistent Programming Systems", Hawaii International Conf. on System Sciences, 1986.

[Atki87a] Atkinson, M.P. and Buneman, O.P., "Types and Persistence in Database Programming Languages", ACM Computing Surveys, 19(2), 105-190, 1987.

[Atki87b] Atkinson, M.P., Buneman, O.P. and Morrison, R., "Delayed Binding and Type Checking in Database Programming Languages", Universities of Glasgow and St. Andrews, PPRR 45, 1987.

[Atki87c] Atkinson, M.P., Lucking, R.R., Morrison, R. and Pratten, D.G., "Persistent Information Space Architecture", Universities of Glasgow and St. Andrews, PPRR 47, 1987.

[Atki88] Atkinson, M.P., Morrison, R., Lucking, R.R. and Pratten, D.G., "PISA Club Rules and Reference Model", Universities of St. Andrews and Glasgow, PPRR 70, 1988.

[Atki89] Atkinson, M.P., "Questioning Persistent Types", Proc. 2nd International Workshop on Database Programming Languages, Oregon, 2-24, June 1989.

[Astr] Astrahan, M.M. et al., "System R: A Relational Approach to Data Management", ACM Trans. on Database Systems 1(2), pp. 97 - 137, 1976.

[Banc] Bancilhon, F., Briggs, T., Khoshafian, S. and Valduriez, P., "FAD: A Powerful and Simple Database Language", Proc. 13th Int. Conf. on Very Large Databases, Brighton, England, pp. 97 - 106, 1987.

[Baye] Bayer, R. and McCreight, E.M., "Organization and Maintenance of Large Ordered Indices", Acta Informatica 1(3), pp. 173-189, 1972.

[Beck] Beck, L.L., "A Security Mechanism for Statistical Databases", ACM Trans. on Database Systems 5(3), pp. 316 - 338, 1980.

[Benz88] Benzaken, V. and Delobel, C., "Clustering Objects on Disk in an Object-Oriented Database", Technical Report, Altair, 1988.

[Benz90] Benzaken, V. and Delobel, C., "Enhancing Performance in a Persistent Object Store: Clustering Strategies in O_2", Proc. 4th Int. Workshop on Persistent Object Systems, Martha's Vineyard, pp. 375 - 384, 1990.

[Berm89] Berman, S., "Persistent Sets - An Alternative to Relational Databases", Proc. 5th South African Computer Symposium, pp. 157 - 168, Johannesburg, 1989.

[Berm91] Berman, S. and von Solms, S.H., "A Formal Model of Persistent Languages based on Subroutine Automata", in preparation.

[Blas] Blasgen, M.W. and Eswaran, K.P., "Storage and Access in Relational Data Bases", IBM Sys. Jnl. 16(4), pp. 363 - 377, 1977.

[Brow85] Brown, A.L. and Cockshott, W.P., "The CPOMS Persistent Object Management System", Universities of Glasgow and St Andrews Persistent Programming Research Report 13-85, 1985.

[Brow88] Brown, A.L., Connor, R.C.H., Carrick, R., Dearle, A. and Morrison, R., "The Persistent Abstract Machine", Universities of St Andrews and Glasgow PPRR 59, 1988 .

[Brow89] Brown, A.L. (PhD Thesis), "Persistent Object Stores", PPRR-71, Scotland 1989.

[Brow90] Brown, A.L., Dearle, A., Morrison, R., Munro, D.S. and Rosenberg, J. "A Layered Persistent Architecture for Napier88", Int. Workshop on Computer Architectures to Support Security and Persistence, Universitat Bremen, May 1990.

[Camp] Campin, J. and Atkinson, M.P., "A Persistent Store Garbage Collector with Statistical Facilities", Persistent Programming Research Report 29, Universities of Glasgow and St. Andrews, 1985.

[Card83] Cardelli, L., "The Functional Abstract Machine", Polymorphism (The ML/LCF/Hope Newsletter) 1(1) 1983.

[Card85a] Cardelli, L., "Amber", Technical Report AT7T. Bell Labs. Murray Hill, USA, 1985.

[Card85b] Cardelli, L. and Wegner, P., "On Understanding Types, Data Abstraction and Polymorphism", ACM Computing Surveys 17(4), 471 - 523, 1985.

[Card88a] Cardelli, L., "A Preview of the Programming Language Quest", 1988.

[Card88b] Cardelli, L., Donahue, J., Kalsow, W. and Nelson, G., "Modula-3 Report", Olivetti Research Centre, Palo Alto USA, 1988.

[Carr] Carrick, R., Cole, J. and Morrison, R., "PS-Algol Reference Manual", Fourth Edition, PPRR 31, Scotland, 1987.

[Chan] Chang, E. and Katz, R., "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS", Proc. ACM SIGMOD Conf., June 1989.

[Chen] Chen, P.P., "The Entity-Relationship Model - Toward a Unified View of Data", ACM Trans. on Database Systems, 1(1) pp. 9 - 36 1976.

[Chin] Chin, F.Y., "Statistical Database Design", ACM Trans. on Database Systems 6(1), pp. 113-139, 1981.

[Codd] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", Comm. ACM 13(6), pp. 377-387, 1970.

[Cock] Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. and Morrison, R., "Persistent Objects Management System", Software Practice and Experience, 14(1), 1984.

[Cole] Cole, A.J. and Morrison, R., "An Introduction to Programming with S-Algol", Cambridge University Press, 1982.

[Conn88] Connor, R.C.H., "The Napier Type Checking Module", Universities of St Andrews and Glasgow PPRR 58-88, 1988.

[Conn89a] Connor, R.C.H., Brown, A.L., Carrick, R., Dearle, A. and Morrison, R., "Design Issues in The Persistent Abstract Machine", Universities of St Andrews and Glasgow PPRR 75, 1989.

[Conn89b] Connor, R.C.H., Brown, A.L., Carrick, R., Dearle, A. and Morrison, R., "The Persistent Abstract Machine", Proc. 3rd Int. Workshop on Persistent Object Systems: Their Design, Implementation and Use, Newcastle Australia, 1989, pp. 80 - 95.

[Conn90] Connor, R.C.H., Brown, A.L., Cutts, Q., Dearle, A., Morrison, R. and Rosenberg, J., "Type Equivalence Checking in Persistent Object Systems", Proc. 4th Int. Workshop on Persistent Object Systems, pp. 151-164, 1990.

[Coop] Cooper, R.L. (PhD Thesis), "On the Utilisation of Persistent Programming Environments", Departmental Report CSC 90/R12, Department of Computing Science, University of Glasgow, April 1990.

[Danf] Danforth, S., Khoshaifan, S. and Valduriez, P., "FAD A Database Programming Language", Technical Report ACA-ST-151-85 Rev. 3, MCC, 1989, to appear in ACM Trans. on Database Systems.

[Date] Date, C.J., "An Introduction to Database Systems", Volume 1, 4th edition, Addison-Wesley, 1986.

[Davi] Davie, A.J.T. and McNally, D.J., "STAPLE User's Manual", Research Report CS/90/12, Department of Computational Science, University of St. Andrews, Scotland, 1990.

[Dear87] Dearle, A., "A Persistent Architecture Intermediate Language", University of Glasgow and St Andrews PPRR 35, 1987.

[Dear88] Dearle, A. (PhD Thesis), "On the Construction of Persistent Programming Environments", University of St Andrews, 1988.

[Dear89] Dearle, A. Connor, R., Brown, F. and Morrison, R., "Napier88 - A Database Programming Language?", in Proc. 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon, June 1989.

[Denn78] Denning, D.E., "A Review of Research on Statistical Database Security", in "Foundations of Secure Computation", DeMillo, R.A., Dobkin, D.P., Jones, A.K. and Lipton, R.J (editors), Academic Press, New York, 1978.

[Denn79a] Denning, D.E., Denning, P.J. and Schwartz, M.D., "The Tracker: A Threat to Statistical Database Security", ACM Trans. on Database Systems 4(1), pp. 76 - 96, 1979.

[Denn79b] Denning, D.E. and Denning, P.J., "Data Security", ACM Computing Surveys 11(3), 1979.

[Denn80] Denning, and Schlorer, J., "A Fast Procedure for Finding a Tracker in a Statistical Database", ACM Trans. on Database Systems 5(1), pp. 88 - 102, 1980.

[DMS] DMS-1100 Database Designers Manual, Sperry-Univac Corporation.

[Ever] Evered, M., "Leibniz - a Language to Support Software Engineering", Ph.D Thesis, Tech. Univ. Darmstadt, 1985.

[Fern] Fernandez, E.B., Summers, R.C. and Wood, C., "Database Security and Integrity", Addison-Wesley, Reading Mass., 1980.

[Gray] Gray, P.M.D, Moffat, D.S. and Du Boulay, J.B.H., "Persistent Prolog: A Searching Storage Manager for Prolog", in Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors), "Data Types and Persistence", Springer-Verlag, pp. 353 - 368, 1988.

[Habb] Habberman, A.N., "Critical Comments on the Programming Language Pascal", Acta Informatica vol. 3, pp. 47 - 57, 1974.

[Hamm] Hammer,M. and McLeod, D., "Database Description with SDM: A Semantic Database Model", ACM Trans. on Database Systems 6(3), pp. 351 - 386, 1981.

[Henn] Hennesy, J. and Elmquist, H., "The Design and Implementation of Parametric Types in Pascal", Software Practice and Experience vol. 12, pp. 169-184, 1982.

[Horo] Horowitz, E. and Kemper, A., "AdaRel: A Relational Extension of Ada", Tech. Report TR-83-218, U.S.C., Los Angeles, 1983.

[Hsia] Hsiao, D.K., Kerr, D.S. and Madnick, S.E., "Privacy and Security of Data Communications and Data Bases", Proc. Int. Conf. on Very Large Databases, pp. 55 - 67, 1978.

[Hugh] Hughes, J.G. and Connolly, M., "A Portable Implementation of Modular Multiprocessing Database Programming Language", Software Practice and Experience 17(8), pp. 533-546, 1987.

[Ichb] Ichbiah et al., Reference Manual for the Ada Programming Language, United States Department of Defence, ANSI/MIL-STD-1815A-1983, 1983.

[Jark] Jarke, M. and Koch, J., "Query Optimization ion Database Systems", ACM Computing Surveys, 16(2), pp. 111-152, 1984.

[Jens] Jensen, K. and Wirth, N., "Pascal User Manual and Report, 3rd edition, ISO Pascal Standard", Springer-Verlag, 1974.

[Kent] Kent, W., "Limitations of Record-Based Information Models", ACM Trans. on Database Systems 4(1), pp. 107-131, 1979.

[Kers] Kersten, M.L. and Wasserman, A.I., "The Architecture of the PLAIN Data Base Handler", Software Practice and Experience, vol. 11, pp. 175-186, 1981.

[King] King, R. and McLeod, D., "A Unified Model and Methodology for Conceptual Database Design", in "On Conceptual Modelling", Brodie, M.L., Mylopoulos, J. and Schmidt, J.W. (editors), Springer-Verlag, 1986.

[Knob] Knobe, B. and Yuval, G., "Towards Pascal II", The Hebrew University of Jerusalem, 1974.

[Koch] Koch, J., Mall, M., Putfarken, P., Reimer, M., Schmidt, J.W. and Zehnder, C.A., "Modula/R Report, Lilith Version", Tech. Report, Institut fur Informatik, ETH Zurich, 1983.

[Leca] Lecarme, O. and Desjardins, P., "More Comments on the Programming Language Pascal", Acta Informatica, vol. 4, pp. 231-243, 1975.

[Lehm] Lehmann, P.L. and Yao, S.B., "Efficient Locking for COncurrent Operations on B-Trees", ACM Trans. on Database Systems 6(4), pp. 650 - 670, 1981.

[Lemp] Lempel, A. "Cryptology in Transition", ACM Computing Surveys 11(4), 1979.

[Lisk] Liskov, B., "Overview of the Argus Language and System", Programming Methodology Group Memo 40, Laboratory of Computer Science, MIT, Cambridge Ma., February 1984.

[Matt85] Matthews, D.C.J., "Poly Manual", Univ. of Cambridge Tech. Report 65, 1985.

[Matt89] Matthes, F. and Schmidt, J.W., "The Type System of DBPL", in Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon, 255-260, June 1989.

[Maie] Maier, D., Stein, J., Otis, A. and Purdy, A., "Development of an Object-Oriented DBMS", Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 472-482, 1986.

[McNa] McNally, D.J., Joosten, S. and Davie, A.J.T., "Persistent Functional Programming", Proc. 4th International Workshop on Persistent Object Systems, Martha's Vineyard, 59 - 70, 1990.

[Miln] Milner, R., "A Proposal for Standard ML", Univ. of Edinburgh CSR - 157 - 83, 1983.

[Morr87] Morrison, R., Brown, A.L., Connor, R. and Dearle, A., "Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment", Persistent Programming Research Report 32, Universities of Glasgow and St. Andrews, 1987.

[Morr89a] Morrison, R., Brown, A.L., Connor, R. and Dearle, A., "The Napier88 Reference Manual", Universities of Glasgow and St Andrews Persistent Programming Research Report 77-89, 1989.

[Morr89b] Morrison, R., Brown, A., Carrick, R., Connor, R., Dearle, A. and Atkinson, M.P., "The Napier Type System", Proc. of the Workshop on Persistent Object Systems, Newcastle, Australia, 253 - 270, January 1989.

[Morr90] Morrison, R., Brown, A.L., Carrick, R., Connor, R. and Dearle, A., "The Persistent Abstract Machine", in "Persistent Object Systems", Workshop in Computing, Rosenberg, J. and Koch, D. (eds.), 279 - 288, Springer-Verlag, 1990.

[Mylo] Mylopoulos, J., Bernstein, P.A. and Wong, H.K.T., "A Language Facility for Designing Database Intensive Applications", ACM Trans. on Database Systems 5(2), 1980.

[Ohor] Ohori, A., Buneman, O.P. and Breazu-Tannen, V., "Database Programming in Machiavelli", Proc. ACM SIGMOD Int. Conf. on Management of Data, Portland, pp. 423-433, 1989.

[Owos] Owoso, G.O., "Data Description and Manipulation in Persistent Programming Languages", Ph.D Thesis, University of Edinburgh, 1984.

[Parn] Parnas, D.L., "On The Criteria to be Used in Decomposing Systems into Modules", Comm. ACM 15(12), pp. 1053-1058, 1972.

[PAM] "The PS-Algol Abstract Machine Manual", University of Edinburgh and University of St Andrews, Persistent Programming Research Report PPRR-11-85, 1983.

[POMS] "The Persistent Object Management System", University of Edinburgh and University of St Andrews, PPR-1-83, 1983.

[Riem] Riemer, M., "Implementation of the Database Programming Language Modula/R on the Personal Computer Lilith", Software Practice and Experience, 14(10), 945 - 956, 1984.

[Rich89a] Richardson, J.E., "E: A Persistent Systems Implementation Language", Ph.D. Dissertation, University of Wisconsin, Madison, August 1989.

[Rich89b] Richardson, J.E., Carey, M. and Schuh, D., "The Design of the E Programming Language", Technical Report No. 824, Computer Sciences Department, University of Wisconsin, 1989.

[Shan] Shannon, K. and Snodgrass, R., "Semantic Clustering", 4th Int. Workshop on Persistent Object Systems, Martha's Vineyard, pp. 361 - 374, 1990.

[Shop] Shopiro, J.E., "THESEUS - A Programming Language for Relational Databases", ACM Trans. on Database Systems 4(4), 1979.

[Smit75] Smith, J.M. and Chang, P.Y., "Optimizing the Performance of a Relational Database Interface", Comm. ACM 18(10), pp. 568 - 579, 1975.

[Smit77] Smith, J.M. and Smith, D.C.P., "Database Abstractions: Aggregation and Generalization", ACM Trans. on Database Systems, 2(2), pp. 105-134, 1977.

[Smit83] Smith, J.M., Fox, S. and Landers, T., "Adaplex: Rationale and Reference Manual - 2nd Edition", CCA, Cambridge Mass., 1983.

[Ston76] Stonebraker, M., Wong, E., Kreps, P and Held, G.D., "The Design and Implementation of INGRES", ACM Trans. on Database Systems 1(3), pp. 189-222, 1976.

[Ston80] Stonebraker, M., "Retrospection on a Database System", ACM Trans. on Database Systems 5(2), pp. 225-240, 1980.

[Tarr89] Tarr, P.L., Wileden, J.C. and Wolf, A.L., "A Different Tack to Providing Persistence in A Language", In Proc. 2nd International Workshop on Database Programming Languages, 41 - 60, 1989.

[Tarr90] Tarr, P.L., Wileden, J.C. and Clarke, L.A., "Extending and Limiting PGraphite-style Persistence", 4th Int. Workshop on Persistent Object Systems, Martha's Vineyard, pp. 71 - 83, 1990.

[Tenn78] Tennent, R.D., "Another Look at Type Compatibility in Pascal", Software Practice and Experience, vol. 8, pp. 429 - 437, 1978.

[Tenn81] Tennent, R.D., "Language Design Methods Based on Semantic Principles", Acta Informatica, vol. 8, pp. 97 - 112, 1977.

[Rose85] Rosenberg, J. and Abramson, D.A., "MONADS-PC: A Capability-Based Workstation to Support Software Engineering", Proc. 18th Hawaii Int. Conf. on System Sciences, pp. 222-231, 1985.

[Rose89] Rosenberg, J., "Pascal/M - A Pascal Extension Supporting Orthogonal Persistence", Tech. Report, Univ. of Newcastle, Australia, 1989.

[Rowe] Rowe, L.A. and Shones, K.A., "Data Abstractions, Views and Updates in Rigel", Proc. of the ACM SIGMOD Conf. on Management of Data, Boston, 71 - 81, 1979.

[Schk] Schkolnick, M., "A Clustering Algorithm for Hierarchical Structures", ACM Trans. on Database Systems 2(1) 1977.

[Saje] Sajeev, A.S.M., "Language Constructs for Persistent Object Based Programming", Proc. 7th IEEE Conf. on Comp. and Comm., Scottsdale Arizona, 1988.

[Schm77] Schmidt, J.W., "Some High Level Language Constructs for Data of Type Relation", ACM TODS 2(3), 1977.

[Schm88a] Schmidt, J.W., Bittner, M., Eckhardt, H., Klein, H. and Matthes, F., "DBPL$^+$ Sysetm: The Prototype and its Architecture", DBPL-Memo 111-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universitat, Frankfurt, 1988.

[Schm88b] Schmidt, J.W., Eckhardt, H. and Matthes, F., "DBPL Report", DBPL-Memo 112-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universitat, Frankfurt, 1988.

[Schm90] Schmidt, J.W., and Matthes, F., "Naming Schemes and Name Space Management in the DBPL Persistent Storage System", Proc. 4th Int. Workshop on Persistent Object Systems, Martha's Vineyard, pp. 39 - 58, 1990.

[Seli] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A. and Price, T.G., "Access Path Selection in a Relational Database Management System", Res. Report RJ2429(32240)/1/8/79, IBM Research Division, San Jose CA, 1979.

[Tenn83] Tennent, R.D., "Some Proposals for Improving Pascal", Computer Languages, vol. 8(3), pp. 125-137, 1983.

[Wai] Wai, F., "Distributed Concurrent Persistent Programming Languages: An Experimental Design and Implementation", Ph.D Thesis, University of Glasgow, 1988.

[Wass] Wasserman, A.I., Shertz, D.D., Kersten, M.L., Riet, R.P. and van der Dippe, M.D., "Revised Report on the Programming Language PLAIN", ACM SIGPLAN Notices, 1981.

[Wels] Welsh, J., Sneeringer, W.J. and Hoare, C.A.R., "Ambiguities and Insecurities in Pascal", Software Practice and Experience, vol. 7, pp. 685-696, 1977.

[Wile] Wileden, J.C., Wolf, A., Fisher, C.D. and Tarr, P.L., "PGraphite: An Experiment in Persistent Typed Object Management", Proc. SIGSOFT '88: Third Symposium on Software Development Environments, pp. 130 -142, 1988.

[Wirt71a] Wirth, N., "The Programming Language Pascal", Acta Informatica vol. 1, pp. 35 - 63, 1971.

[Wirt71b] Wirth, N., "The Design of a Pascal Compiler", Software Practice and Experience, vol. 1, pp. 309 - 333, 1971.

[Wirt88] Wirth, N., "The Programming Language Oberon", Software Practice and Experience 18(7), pp. 671-690, 1988.

[Wood] Wood, C., Fernandez, E.B. and Summers, R.C., "Data Bas Security: Requirements, Policies and Models", IBM Sys. Jnl. 19(2), 1980.

[Yu77] Yu, C.T. and Chin, F.Y., "A Study on the Protection of Statistical Databases", ACM SIGMOD Int. Symposium on Management of Data, pp. 169 - 181, 1977.

[Yu85] Yu, C., Lam, K., Siu, M and Suen, C., "Adaptive Record Clustering", ACM Trans. on Database Systems 10(2) 1985.