University of Glasgow
Department of Computing Science

Lilybank Gardens
Glasgow G12 8QQ

University of St Andrews
Department of Computational Science

North Haugh
St. Andrews KY16 8SX

DATABASE PROGRAMMING LANGUAGE DESIGN

Persistent Programming
Research Project 17

Alan Dearle.

Database Programming Language Design

Malcolm P. Atkinson and O. Peter Buneman

UNIVERSITY of GLASGOW
Department of Computing Science
GLASGOW G12 8QQ
Scotland

UNIVERSITY of PENNSYLVANIA
Department of Computing and Information Science
Philadelphia, PA 19104, USA

This work was carried out at the Universities of Glasgow and St. Andrews and the University of Pennsylvania.

Abstract

The necessity of integrating database and programming language techniques has recently received some long-overdue recognition. This paper reviews a number of attempts to construct *database programming languages* by which we mean programming languages which incorporate some kind of database technology. Traditionally the interface between a programming language and a database has been through a set of relatively low-level subroutine calls. More recently, a number of attempts have been made to construct programming languages with completely integrated database management systems. Between these two extremes lie a number of other systems whose purpose is to ease the burden of writing database applications in high-level languages.

The design of these languages is still in its infancy; and the purpose of writing such a review is to attempt to identify those areas in which further research is required. In particular we shall focus on the problems of providing a uniform type system and providing mechnisms for data to persist. Other areas, of equal importance, such as transaction handling and concurrency are not examined here in any detail.

Note to readers: In reviewing a subject that is changing so rapidly, it is impossible to keep abreast of all developments; and while we have attempted a comprehensive survey there are undoubtedly ommissions and innaccuracies. We would be grateful for any information on new languages or revisions and developments of those we discuss so that these can be included subsequent versions of the paper. We would be interested to learn of better solutions to the problems that we have used to illustrate the languages.

Categories and Subject Descriptors:
S.1.4 [**Programming Techniques**]:
Sequential programming, Functional programming;
D.2.2 [**Software Engineering**]: Tools and techniques;
D.3.3 [**Programming Languages**]: Language constructs.

Keywords: Data bases, Data types, Programming languages,
Persistence, Polymorphism, Data models
Conceptual languages, Embedded languages,
Integrated languages, Persistent languages,

1. Introduction

Developments in databases and programming languages have proceeded almost independently of one another for the past twenty years. Since nearly every program we write requires access to some form of "permanent" data, enormous manpower has been expended on developing individual databases for specific applications or in using a programming language - database interface that requires considerable expertise. Worse still are the impediments placed on system development by the lack of adequate programming tools for databases. It frequently happens that the presence of an existing database, maintained under an established database management system has put an effective stop to the use of advanced programming tools simply because the interfaces do not exist and because it would be operationally impossible to restructure the data for a better programming environment. As a result there are organizations that cannot adapt to changing needs because of these limits to system development.

Until quite recently, the picture in database and programming language research was equally discouraging. Both subjects had accumulated a few good prototypes; so that research in the separate areas could be fairly easily classified by reference to one of a limited family of languages or data models. Only recently has the need to produce an integrated system for programming and data management been recognised, and this has led to a number of attempts to produce such systems either by writing a completely new database programming language or by enhancing an existing language with some form of database management. In either case the usual approach has been to combine an existing language with an existing data model.

Our purpose in writing this paper is to give an account of these attempts to integrate programming languages and data bases in the hope of extracting certain principles that should be used in future attempts to design programming languages or databases. In fact, it is our hope that all future attempts at research in either area will attempt to develop a unified solution. Such a unified solution needs some guiding principles and should take note of what has already been attempted. We therefore want to present a survey of past and current efforts with reference to some test cases.

The three principles we shall elicit are *persistence*, *type completeness* and *computational power*. To a limited extent, each of these can be given some formal, semantic, definition; however, much theory remains to be developed. We are more concerned here with the practical consequences of these principles and will attempt to

clarify them and justify their importance through examples.

To illustrate *persistence*, imagine a second-level programming course in which students are taught to write B-tree type declarations and manipulation routines in, say, Pascal. This well-known excercise is an excellent illustration of how Pascal's data types may be used both in formulating the problem and in preventing the kinds of error that plague pointer manipulation. Unfortunately, the student's code serves only as a model for the code that is involved in a real implementation. It cannot be used to exploit secondary storage - the *raison d'etre* of B-trees - since the only persistent data type in Pascal, the file, will not accomodate this structure. To implement a working B-tree, the student will have to resort to a lower-level language for implementation or play tricks on the Pascal compiler that violate the type rules, in both cases leaving open again the possibility of pointer errors.

Type completeness refers to the way in which all data types should enjoy equal status within a language. A well-known example of failure of type completeness is again to be found in Pascal in which functions can only return values of a limited set of data types. Another well recognised problem with Pascal's types is the impossibility of expressing a new *parameterised* type. The student's B-tree code should surely work for an arbitrary index type and an arbitrary result type, yet if we use Pascal's type system strictly, new lookup and update procedures must be generated for every pair of types.[1] A more serious problem will be seen with many of the languages in this paper in which only certain data types (such as files or relations) have the right to persist. For the most part, persistence and type completeness go hand-in-hand. They are however separate issues.

The lack of *computational power* is well known to anyone who has used a database query language. There are programs, often involving numerical computations or complicated manipulation of the database, that cannot be written in the language. The traditional solution to this problem has been to use two languages, sometimes embedding the query language in a computationally more powerful "lower-level" language. While nearly all the programming languages we shall investigate provide adequate computational power, they have often been embellished with new programming constructs for manipulating databases. The uniformity with which these constructs

[1] It should be stated that we are using, and criticizing, Pascal for purely pedagogic reasons. It is precisely because the language achieved so much in making practical use of data types that we are in a position to discuss the next steps.

apply to other data types is then the issue. In fact, computational and type completeness are closely related; while persistence is again an independent issue.

We have written this paper as a review of a number of languages which, we hope, will provide ideas for future language designers. We have chosen certain languages because we believe they have made contributions to the development of database programming languages (DBPLs). The fact that we are often critical of these languages should not detract from their importance. They were chosen because, although programming language research is extensive, only a handful of language designers have recognised the importance of database programming. There are undoubtedly omissions and the authors would welcome information on other systems that are being developed.

Finally, a word about our methods. After elaborating on some of these concepts we shall present the languages through a number of worked examples. The reader may get bored with the detail and - quite reasonably - want to skip whole sections. This is especially true of the first sections in which we have tried, through various examples, to explain the problems for the benefit of both the database and programming language communities. We hope the accompanying text makes it easier for the reader both to understand the examples and perhaps to experiment with alternative designs for a DBPL.

1.1. Provision of Independent Persistence

In procedural languages, we normally think of objects as having a well-defined lifetime. A variable declared in a block or procedure will "persist" during the activation of that segment of code and thereafter will be inaccessible. If an object is created as part of a data structure, its persistence, from the user's point of view, is the duration for which it remains accessible. Some languages allow the explicit (and dangerous) control of persistence through the use of storage de-allocation procedures such as **free** or **dispose**. Without explicit deallocation, the logical persistence of objects during program execution is well-understood. Moreover these mechanisms (scope rules, allocation and de-allocation) can be apply to a wide range of types.

However, when data are required to last longer than the duration of one program execution, the treatment is much less regular. For example the set of types that may persist from one program execution to the next is often a small subset of, or even different from, the types available in the shorter time scales. In most languages the only

objects endowed with long-term persistence are *files*; and while, in Pascal, files may parameterised by other types and therefore can be used as a vehicle for the persistenc other objects, the parameterization is incomplete and cannot be used on, for exam pointer types. Thus the programmer is constrained either to use a possi unsatisfactory subset of the available types, or must resort to "loopholes" through wh the physical representation of types may be manipulated. The dangers of the lat option need no further elaboration here.

We have argued elsewhere that this discontinuity in the treatment of data associat with a change in persistence is both deleterious and avoidable [Atkinson *et* 83a, Atkinson *et al.* 84, Atkinson & Morrison 84]. The languages presented here displ the discontinuity to a varying degree: the well-known programming languages, mere support a very small subset of their types as persistent;[2] some languages provide different sublanguage for their long term data storage; others introduce new types f data storage; whilst some extend the range of persistence beyond program execution b having a mechanism for storing their workspaces. Generally, in the languages w examine, we find persistence presented as a binary division between near permanenc and transient existence though we suspect that intermediate forms may be useful.

Although we cannot yet give precise semantics for persistence, we hope to convince th reader that there are certain principles of persistence that should govern languag design:

1. Persistence should be a property of objects, not of types.

2. All objects should have the same rights to persistence.

3. While an object persists, so should its description (type).

The first two of these state that an object's persistence should be independent of its type; persistence should therefore be regarded as a property of data orthogonal to its type. A corollary is that the code used to manipulate an object should not depend on its persistence. Failure to comply with the third principle is a common source of system error. It should not be possible to write out an object as one type and subsequently to read it in as another. This may be regarded more as a property of program environments rather than programming languages. We believe that for data base programing languages the issues of programming language and programming

[2]This subset is often further limited by the language's implementation.

environment cannot be separated. Indeed, one of the authors [Atkinson and Morrison 84]has recently demonstrated that program linking - normally associated with programming environments - can be neatly represented by persistent functions.

A language in which the programmer has to include explicit statements to initiate or organise transfer of data objects would not comply with the requirement that the code to manipulate an object should not depend on its persistence. The required transfers between stores can, and should, be inferred from the operations on the data. A programmer is distracted from the essence of an algorithm when reading or inserting explicit transfer statements.

A counter argument to providing persistence is that it is difficult to find good engineering techniques to support arbitrary persistent structures. Certainly, possibly because of the research effort expended, the mechanisms for some types, such as those constructed as relations are better understood at present.

1.2. Data Type Completeness

Data Type Completeness is only meaningful in languages that offer some form of type parameterisation. For example, if the language allows an **array of** *a* construct, then we should be able to use this with any type value for *a*. Thus **array of** *char*, **array of array of record** ..., etc. should all be permissible types. Since procedures and functions may themselves be regarded as having types (determined by the types of their arguments and results), completeness demands that any type should be admissible as the value of such type parameters, including procedure or function types.

The concept recieved a boost with its practical demonstration in the languages of the early seventies, especially Algol68 [van Wijngaarden *et al.* 69] and Pascal [Wirth 71]. They demonstrated that it was feasible to specify languages with a high level of data type completeness, and then to implement such languages. However, engineering considerations have tended to deter complete adherence to the principle. For example **set of** in Pascal is limited to scalar types; one cannot have **set of array...**, **set of record...** etc. As another example, arrays, records, sets and functions may not be the result of a function.

While languages in the Algol-Pascal family allow the user to construct new data types by use of predefined type constructors - types such as **array** and **file**, which when

"applied" to a type yield a new type - they do not allow the programmer to define type constructors such as *list*, *stack* etc which require a type parameter. Thus a programmer who needs to build several different types of linked list must replicate the type definitions for each type of element, a consequent problem is that the same procedures (*cons*, *reverse* etc.) for each type of list must be written. A more recent development has been the introduction of languages (ADA [Ichbiah *et al.* 79], ML [Gordon *et al.* 79], POLY [Matthews 85a], RUSSEL [Demers & Donahue 79] and PONDER [Fairbairn 82, Fairbairn 85]) which, in various ways, allow user defined parameterised types. In these languages the user may define a data type *list_of* and use it with different parameters: *list_of* **record** ..., *list_of* **integer** etc. In ML, for example, an expression α *list* (ML uses reverse notation for types) denotes a type for any type α. The term *parametric polymorphism* is used to describe this form of parameterization. In other languages [Ichbiah *et al.* 79, Demers & Donahue 79, Matthews 85a] one may define a type *sorted_list* α, which is valid only if the type α satisfies certain conditions (e.g. there is a comparison function $\leq$ defined for it). It is also important to distinguish between *overloading* and *parameterization*. The former refers to the ability of a name to refer to different objects depending upon its type, which is usually inferred from context. For example, the implementation and behaviour of + depends upon whether its arguments are reals, integers or (possibly) character strings. Despite its importance, type parameterization is not yet widely understood, nor has it been developed to cover all the requirements of database programming languages. An excellent survey has been prepared recently [Cardelli & Wegner 85].

The data type declarations in a program perform a very similar function to the data description language in a database schema, and we will be concerned with their relative capabilities. Type declarations can serve to describe named classes of data structures. In the case of abstract data types, they introduce a set of names and associated specifications for their use. These specifications can be formally described by an algebraic *signature*. Where both data description languages and type declarations exist separately the database interface needs an arrangement for matching the names and associated constructs. In early interfaces this was quite cumbersome but mechanisms to simplify the import of database names into the program have been developed progressively. However, semantic differences remain to trap the unwary. The set of objects of the "types" provided by the DDL invariably persist longer than the execution time of a program, but methods of providing transient instances of these types may not

be supported. Similarly, the instances of the programming language's types may have transient persistance (i.e. within the execution time of one program) but may not be allowed longer persistence. This failure of independent persistance, invariably results in a failure of type completeness.

At present there are marked differences between the capabilities of the type regimes in the programming languages, and those that we would like if we were to be able to declare all the types introduced by the data models in their DDLs. We can view a given data model as a type (type constructor) which when parameterised (with a schema text) yields a new type. All databases described by that schema are then instances of that type. Although it is not, at present, possible to write down parameterised types to correspond to each data model in any programming language yet developed, this is such an important path towards unification that we explore the language trends which make this look potentially possible. These include the polymorphic and parameterised types introduced above.

1.3. Computational Power

To a programming language designer it is unthinkable that a language should not be able to perform abitrary computations; but in database query languages there are usually severe limits to the kinds of computations that can be performed. We shall examine in this paper an example of a common class of computation that defeats most query languages. Another failure of many of these languages is that they cannot evaluate, at the top level, simple arithmetic expressions even though such expressions may be permissible as part of a larger relational expression.

There are several arguments that can, and have, been given in favour of restricting the power of a query language. One is that it simplifies the language for the user, who is not usually a programming language expert. It is certainly true in the languages that we shall survey that the languages that allow arbitrary computations are considerably more forbidding than the query languages. We do not yet know whether this is necessarily the case for all future database languages. We hope not and are encouraged by the simplicity of logic programming [Clocksin & Mellish 81] and some of the simple applicative languages such as SASL [Turner, D.A. 81] that allow the user to start by evaluating very simple queries but do not impose any limit to the power of computations that can be performed. Another reason for limiting computational power is to provide a subset of operations that can be efficiently implemented on a language that can be effectively

optimised. Again, the limits are arbitrary. While most query languages will compute an average of a set of numbers, few will compute the variance, even though any data base machine or management system that can efficiently compute an average should be able to do the same for a variance. Moreover, many of these languages will not allow the user to define a function that computes the variance.

Since most of the languages we shall deal with are database programming languages as opposed to query languages, they do provide adequate computational power. However, the way this is provided is not always uniform. Several of the languages allow an iterator of the form **for** *x* **in** *S* **do** . . but limit this to cases where *S* is a sequence or set in the database. Whether this is a failure of computational power or type completeness is arguable, nevertheless a shift of compuatational strategy is often forced upon the programmer for the performance of more complicated tasks. This is especially true of embedded languages and languages that are in some sense combinations of a programming language and a database query language.

2. A Test Case and some Basic Approaches

In order to illustrate in a concrete fashion the issues presented in the previous section, we shall use throughout this paper a specific set of database programming examples. In doing this, we must obviously make some compromises. Database schemas can be extremely complicated (hundreds of record types or relations may be involved), and programs can be lengthy, involving widespread interaction with the user and with many parts of the database. Given that space prohibits the presentation of large bodies of code, the database example presented here is intended to suggest realistic components of some large system. The example we shall use is an illustrative *fragment* of a manufacturing company's parts database. The reader is invited to imagine the rest of this database, and the other processes and programs that would use it. The database represents among other things the inventory of a manufacturing company. In particular it represents the way certain parts are manufactured out of other parts: the subparts that are involved in the manufacture of a part, the cost of manufacturing a part from its subparts, the mass increment or decrement that occurs when the subparts are assembled. Note that manufactured parts may themselves be subparts in a further manufacturing process. The relationship between parts is therefore hierarchical, but it is a directed acyclic graph rather than a tree, for part D may be used in the manufacture of parts B and C, which are both used in the manufacture of part A. In addition, certain information must be held on the parts themselves: their name, identifying number and, if they are imported, (i.e. manufactured externally) the supplier and purchase cost. The first task is therefore

Task 1: *Describe the database.*

In traditional terms we are asking for a database schema, however in the languages we shall examine, this description is sometimes part of a type definition. Not all the descriptions will be equivalent (in the sense that they could be automatically transformed one into another). For example, there are conditions on the database that in some data models are implicit, in others, can be represented explicitly in the type declarations (schema definition) or as integrity constraints automatically enforced, whilst in other systems it may be the responsibility of programmers to ensure that all updating programs preserve these conditions.

Having defined the database, we want to write three programs against it. The first is simple and is chosen to provide an introduction to each language:

Task 2: *Print the names, cost and mass of all imported parts that cost more than $100*

Database query languages are designed to make the expression of queries like this extremely simple; and a measure of programming languages in general is the simplicity with which the code for queries such as this can be expressed.

The next example is somewhat more complicated and defeats many query languages:

Task 3: *Print the total mass and total cost of a composite part.*

Since a part may be made from other parts including parts which themselves are composite, this calls for at least one recursive traversal of the parts hierarchy of the database. The inability of relational query languages to perform recursive traversal or "transitive closure" has long been recognised [Aho & Ullman 79].[3] This example poses some additional efficiency problems: avoiding repeated computation of costs and masses of subparts, and the desire to compute both cost and mass in parallel.

Finally we would like to demonstrate update by adding some information to the database:

Task 4: *Record in the database a new manufacturing step, i.e. how a new composite part is manufactured from subparts.*

The point of this last example is to examine where, in the program or type system, integrity constraints are implemented. Implementation of such updates are dominated by user interface construction and by the need to isolate the dialogue with the user from the consequent action of changing the database. In our examples we have largely ignored the user interface, but have shown how the isolation may be achieved. This update task also raises several issues central to databases, but largely ignored in programming languages. How the change is reliably recorded? When does it becomes visible to other users? These questions introduce the topics of concurrency, transactions and protection. Lack of space precludes their detailed discussion in this paper, but their omission is certainly no indication that they are of lesser importance.

As we have already remarked, these examples only serve to represent a small fragment of what would actually be involved. The need to represent other data would better set the context of this example. For instance, it would be desirable to provide details of suppliers, their reliability, current stock, current orders, delivery times, assembly times,

[3] Considerable work has recently been done to extend query languages to include some form of transitive closure, but it is not clear whether these address efficiency issues. Even then, a solution to the transitive closure operation itself does not necessarily solve all the problems of a database interface, in particular it does not solve our third task.

manufacturing processes, part strengths, shapes etc. Similarly many processes would normally use such data, such as design, testing, maintenance, production etc. It is also likely that short examples like these will fail to do justice to certain features of the languages we shall review. While we shall comment on these where possible, we must again excuse ourselves from a detailed presentation on the grounds of lack of space.

To present the technical issues in terms that should be familiar to most people, we shall attempt these tasks in Pascal and a relational database query language. Neither of these languages can be regarded as database programming languages; Pascal has inadequate persistence and most query languages have inadequate computational power. However, many of the attempts to produce database programming languages are based on a fusion of the two approaches. These two examples will therefore serve to introduce the basis for many of the later comparisons.

2.1. The programming language approach illustrated with Pascal

Many applications are written using a programming language alone, either managing without permanent data, or using some programmer contrived mechanism to store the data in files between program runs. Pascal has seriously limited forms of persistent data, so, like the B-tree example cited earlier, our code is no more than an illustration of what needs to be done; but we can use the types of Pascal to provide a representation of the part assembly as it might be implemented in secondary storage. We could, of course, code a real (secondary storage) database in Pascal, but we would have to resort to our own pointer management, and would not gain the benefits of Pascal's type checking.

Figure 2-1 shows a type declaration corresponding to Task 1.[4] The data structure we have chosen is a standard linked-list implementation of a many-many relationship, as described in [Date 83a]. It is also similar to the Codasyl implementation that we shall describe later. Note that the only available "bulk" data types available in Pascal, **array**, **set** and **file** are inappropriate for this task. Moreover the programmer cannot construct an appropriate parameterised data type such as *list_of*. Had this been possible we could then have declared types *list_of Part* etc to model the collective objects in the schema

[4] When we first worked through the various examples, we tried to adopt a uniform type font and case convention for various languages. However, we could not find any convention that was consistent with the cultures of all the languages in which we were interested. Instead, we tried to work each example in the style of the descriptions and reference manuals at our disposal. We have however tried to maintain a uniform naming convention, thus *Part* will usually refer to a data type and *aPart* to a value. Even maintaining a naming convention is not always possible.

```
type
    PPType      =  (Base, Composite);
    String      =  packed array [1..16] of char;
    Grams,
    Dollars     =  integer;

    PartRef     =  ↑Part;
    PartList    =  ↑PartCell;
    UsesList    =  ↑Use;
    UsedInList  =  ↑Use;
    SuppList    =  ↑SuppCell;

    Part        =  record                {to represent a part}
                     Name: String;
                     UsedIn: UsedInList;
                     case PartType: PPType of
                       Composite:  (AssemblyCost: Dollars;
                                    MassIncrement: Grams;
                                    MadeFrom: UsesList);
                       Base:       (UnitPrice: Dollars;
                                    Mass: Grams;
                                    Suppliers: SuppList)
                     end;

    Use         =  record        {represents the assembly-component relationship}
                     Quantity: integer;
                     Uses, UsedIn : PartRef;
                     NextUses: UsesList;
                     NextUsedIn: UsedInList
                   end;

    PartCell    =  record        {to represent the set of all parts}
                     Cont: PartRef;
                     Next: PartList
                   end;

    SuppCell    =  record ... end;

var Database :      record
                      Parts: PartList;
                      Suppliers: Supplist
                    end;
```

Figure 2-1: Task 1: describing the data in Pascal

explicitly. On the positive side, note that the variant record mechanism of Pascal neatly captures the specialization hierarchy of this example, and the requirement that every part be either a *CompositePart* or a *BasePart*. It models this case well as the specializations are *exclusive* but cannot deal so readily with specialization where *overlap* is permitted. Another advantage of Pascal (as opposed, say, to PL/I) is typing of pointers. This is of substantial benefit in writing applications against structures such as these where it is extremely easy to confuse two pointers. We shall see this problem of type checking again in many of the traditional methods of programming against Codasyl databases. There are in fact two kinds of type checking that are used (often simultaneously) in database programming languages: *static* type checking like that in Pascal will simply

insist on all objects being typed in advance of program execution; *dynamic* checking prevents type errors at run-time by raising exceptions. In the implicit pointer manipulation that is performed for some databases, one frequently has neither.

The main omission of this type declaration is that there is no provision for efficient look-up of, say, a *Part* record given its *Name*. We need here some sort of indexing mechanism, such as a B-tree or hash table that would implement an efficient look-up. Apart from the fact that standard Pascal would not allow us to build a *persistent* index structure, there is the additional, and more serious, problem that one could not make it generic. If one were to work within the type framework of Pascal, one would have to write two sets of almost identical routines to implement both *Name* to *Part* table and, say, a *Sname* to *Supplier* table.

```
program Task2;

...                              {The type declarations of fig 2-1}

var pl: PartList;

...                              {Code to build the data base must go here since
                                  we have no persistence}

begin
  pl := Database.Parts;
  while pl ≠ nil do
    begin
      with pl↑.Cont↑ do
        if (PartType = Base) and (UnitPrice > 100) then
          writeln(Name, UnitPrice, Mass);
      pl := pl↑.Next
    end
end.
```

Figure 2-2: Task 2: A Pascal program to retrieve expensive parts

Figure 2-2 presents a Pascal solution to Task 2. It follows immediately from the type declaration.

Figure 2-3 sketches a Pascal solution to Task 3. Organising the recursive traversal of the part hierarchy presents no logical problems. The introduction of parameters and of local variables to represent partial totals permits both values to be calculated in one traversal (something that cannot be done in some other languages). The structure of *costAndMass* is easy to recognise. In contrast the intent of the relatively simple function *findPart* is much less obvious. Again, what is required here is an efficient (search tree or hash table) indexing type. Any DBPL should incorporate a generic index type. A

```
procedure Task3;

...                          {The type declarations of fig 2-1}

function findPart(pn:String):PartRef;
                             {Returns reference to part named pn; nil  otherwise}
var pl: PartList; result: PartRef;
begin
  result := nil;
  pl:=Database.Parts;
  while (result = nil) and (pl ≠ nil) do
    if pl↑.Cont↑.Name = pn then result := pl↑.cont;
    else pl:=pl↑.Next;
  findpart := result
end;

procedure costAndMass(p: Part; var resultCost: Dollars;
                              var resultMass: Grams);
var subTotalCost: Dollars; subTotalMass: Grams;  ul: UseList;
begin
  with p do
    if PartType = Base then
      begin
        resultCost := UnitPrice; resultMass := Mass
      end
    else
      begin {recursively compute costs and masses of subparts}
        resultCost := AssemblyCost; resultMass := MassIncrement;
        ul := MadeFrom;
        while ul ≠ nil do
          with ul↑ do
            begin
            CostAndMass(Uses↑, subTotalCost, subTotalMass);
            resultCost := resultCost + quantity*subTotalCost;
            resultMass := resultMass + quantity*subTotalMass;
            ul := NextUses
            end
      end
end;

procedure main;
var itsCost: Dollars; itsMass: Grams; pref: PartRef;
begin
  pref := findPart('Mast          ');
  if pref ≠ nil then
    begin
      costAndMass(pref↑, itsCost, itsMass);
      writeln(itsCost, itsMass)
    end
end;

begin main end.
```

Figure 2-3: Task 3: Pascal code to compute cost and mass simultaneously

minimal requirement of Pascal and related languages for database programming is the provision of a persistent version of such a type.

The question posed by these examples is whether the powerful higher level types such

as index or relation can be introduced - making Task 2 and *findPart* succinct - without loss of the regularity which makes *costAndMass* straightforward to code. There is also an inefficiency in the code of *costAndMass* concerned with the recalculation of data about common subassemblies, which we will show later can be addressed without obscuring the algorithm if such higher level types are available.

```
program Task4;

...          {The type declarations of fig 2-1}

...          {Code to interact with the user to get the name, assembly cost
              and mass of a new part together with its subparts and
              their quantities. We assume that a PartRef pr has been
              constructed together with a MadeFrom list, but this has not
              yet been installed in the database; i.e. pointers exist from this
              PartRef to structures in the database but not vice versa}


var ul: UsesList; pl: PartList;
begin      {install the part referenced by pr}
  ul := pr↑.MadeFrom;
  while ul ≠ nil do
    with ul↑ do begin {add each component to uses list}
      NextUsedIn := Uses↑.UsedIn;
      Uses↑.UsedIn := ul;
      ul := NextUses
    end;
  new(pl);
  pl↑.Cont := pr; pl↑.Next := Database.Parts; Database.Parts := pl;
end;
```

Figure 2-4: Task 4: Pointer manipulation required to install a part

Figure 2-4 shows how Task 4 may be coded in Pascal. No facilities exist in the language for identifying this as a transaction[5] or for arranging to deal with concurrent access.

2.2. The relational approach

Since its initial formulation [Codd 70] the relational model has been a basis for discussing data design, and, as techniques have been developed for the efficient representation of relations and relational operators, relational systems have become increasingly marketable. [Schmidt & Brodie 83] gives a good survey of this species of database system.

Figure 2-5 shows how the data could be described in a prototypical relational system.

[5] i.e. a collection of operations that appear (a) to run to completion or not to run at all and (b) are indivisible in that no other program can run against the same data during the transaction [Date 83a]

```
Part(Pno: Pnum, Name: String)
BasePart(Pno: Pnum, UnitPrice: Dollars, Mass: Grams)
CompositePart(Pno: Pnum, AssemblyCost: Dollars, MassIncrement: Grams)
MadeFrom(Assembly: Pnum, Component: Pnum, Quantity: PosInt)
SuppliedBy(Pno: Pnum, Sno: Snum)
...
```

Figure 2-5: A DDL fragment for a relational representation of the database

It shows the column and domain names of each of five relations with the primary keys underscored[6]. But it does not define the interdependence of the relations, nor the interpretation of the values within them. Some would contend this is a symptom of a general shortcoming of the basic relational model, in that it fails adequately to define the interrelationships between relations [Date 81b] and carries insufficient indication of the semantics of a database [Codd 79, Kent 78, Kent 79]. In contrast, Merrett contends [Merrett 83] that this lack of semantics is to the relational model's advantage, making it a better basis for organising data storage, and as a kernel for data manipulation languages. He illustrates this with a demonstration of how the model may represent line drawings [Merrett & Düchting 84] and text [Merrett 85a]. It should also be noted that relational systems do not usually provide, or allow the definition of, the domains used in figure 2-5.

Figure 2-6 indicates some of the constraints needed in addition to those imposed by the keys in the relational schema. These have been stated in predicate calculus but could equally well have been stated as equations or inclusion relationships on relations derived using relational algebra. Contrasting this with the Pascal schema (type declaration) we see that the use of reference types can be used to capture some of these constraints. In relational databases these constraints can also be stated in terms of *foreign keys*, which have been discussed [Date 81b, Codd 79] but not, to our knowledge, implemented in any practical relational database management system.

The complexity of programming with pointers was one of the strongest arguments for the use of the relational model. At the time the relational model was introduced, much database programming was done through the use of low-level packages that manipulated pointers on secondary storage. One can imagine what these early database management systems were like if one removes the type information from our Pascal program thereby laying open the possibility of confusing pointers to different types of objects. However,

[6] For tutorial material on relations the reader is directed to one of the following books [Date 81a, Date 83b, Merrett 84, Tsichritzis 77, Ullman 82]


```
{Each base and composite part is also a part}
for each b in BasePart there exists p in Part such that p.Pno = b.Pno
for each c in CompositePart there exists p in Part
                        such that p.Pno = c.Pno

{A part must be a composite part or a bought in part}
for each p in Part
        there exists b in BasePart such that p.Pno = b.Pno
    or
        there exists c in CompositePart such that p.Pno = c.Pno

{Exclusion - we don't make parts we buy}
for each b in BasePart, each c in CompositePart b.Pno ≠ c.Pno

{Only Composite Parts can be assembled}
for each m in MadeFrom there exists c in CompositePart
    such that m.Assembly = c.Pno
    and ((there exists c' in CompositePart such that m.Component = c'.Pno)
    or  (there exists p in Part such that m.Component = p.Pno))

{We only buy-in Base Parts}
for each s in SuppliedBy there exists b in BasePart
                        such that s.Pno = b.Pno

{All base parts have a supplier.}
for each b in BasePart there exists s in SuppliedBy
                        such that b.Pno = s.Pno
...
```

Figure 2-6: Some integrity constraints on the parts relations

the introduction of typed languages and data definition tools (as well as partially typed systems such as Codasyl) has done much to alleviate these problems. Nevertheless our Pascal types are still not nearly as simple as the relational schema. The problem here is that references are used both to maintain *referential integrity* and to build the data structures required in the database. The importance of referential integrity has been supported by [Atkinson 78, Date 81b]. If and when implementations of these models exist, it will be interesting to see how the treatment of references combines with their treatment in a host programming language.

The first two constraints of figure 2-6 also suggest a type hierarchy, which was modelled as a variant record in the previous Pascal example. Later these ideas of generalisation and specialisation heirachies [Smith 77] are seen as fundamental to the semantic data models [Brodie *et al.* 83, Borgida 83]. A number of constraints could have been made unnecessary by eliding the first three relations, this would leave a less precise description of less compact data, and the problems posed above would not have gone away, as they would reappear in the treatment of *not applicable* or *null* values.

Most relational systems offer a simple query language, and data manipulation language.

```
SELECT   Part.Pname, BasePart.UnitPrice, BasePart.Mass
FROM     Part, Basepart
WHERE    Part.Pno=BasePart.Pno
AND      BasePart.UnitPrice ≥ 100
```

Figure 2-7: Task 2: retrieve details of expensive parts in SQL

This would usually suffice for Task 2 as we illustrate by coding the query in SQL [Date 83b] as shown in figure 2-7. This relational calculus query avoids much of the complication of the Pascal program (figure 2-2) as it does not need to organise the data scan, nor to organise output explicitly. However, the Pascal program had one advantage, it did not need a join (the first line of the **WHERE** clause) to deal with the type inheritance implicit in the specialization hierarchy.

By contrast Task 3 is impossible in most query relational languages. In fact it can be shown [Aho & Ullman 79] that a general *transitive closure* operation cannot be expressed with a relational algebra expression. Since relational query languages are based indirectly on relational algebra, and since this task calls for something at least as powerful as a transitive closure operation, the failure of query languages at this level of complexity is not surprising. A user confronted with a problem such as this would therefore be confronted with the following options:

1. Unload the relevant portions of the database into a file and use a separate language, or even hand calculation, to complete the task.

2. Use a more powerful language that can call directly upon the query language for partial computations. We shall discuss this under *embedded* languages.

3. Write a query that performs the computation to some finite depth. For example, one might assume that no assembly has a sub-assembly which itself has sub-assemblies.

The first option is probably the most frequently taken in the commercial world. Analysts or "end users" are unaware of what software may be available to perform more complicated queries, and systems programmers and designers are unaware of the amount of hand calculation that follows from a limited query language or set of reports. This has three consequences: the requirement for more general computational power continues to be underestimated, the volume of data transferred or printed is unnecessarily high, and there is a possibility of introducing errors during transcription and interpretation.

Both of the first two options require the user to undertstand a separate and more complicated programming system and place an effective barrier on the sophistication of queries that users can pose for themselves. The difficulty of acquiring skills in the more

complicated language - due more to lack of time rather than lack of intellect - often account for the rather artificial distinction between "applications" and "systems" programmers, and accounts for the common frustrations of data analysts who do not see themselves as computer programmers.

```
Rel1(Pnum, Vc, Vm) <-
        SELECT Component.Pno, Quantity*Unitprice, Quantity*Mass
        FROM Composite, Base, Use
        WHERE Composite.Pno = Use.Assembly
        AND Base.Pno = Use.Component

Rel2(Pno, Vc, Vm) <-
        SELECT Pno, AssemblyCost, MassIncrement
        FROM Composite

Rel3(Pno, Xc, Xm) <-
        SELECT Pno, SUM Vc, SUM Vm
        FROM
        (SELECT *
         FROM Rel2 )
        UNION
         (SELECT *
        FROM Rel3)
        GROUP BY PNum

Rel4(Pno, Xc, Xm)
        <- SELECT Pno, UnitPrice, Mass
        FROM Base

Answer(Pno, itsCost, itsMass) <-
        (SELECT *
         FROM Rel3)
        UNION
        (SELECT *
         FROM Rel4)
```

Figure 2-8: A partial result for Task 3 in SQL

Figure 2-8 shows an attempt at the third option on the very unrealistic assumption that the parts explosion diagram is only one deep, i.e. composite parts are only constructed from base parts. The solution presented here may not be the shortest or the most efficient. The computation proceeds by first constructing a relation, *Rel1* that contains part numbers of each composite part and the total cost and the total mass for each of its subparts (which must be in *Base*). *Rel2* essentially renames *Composite* so that its column names are compatible with those of *Rel1*. *Rel3* is then constructed to contain the total cost and total mass of each composite part. By further relabelling of columns, we include the base parts with their costs and masses to get the final result, a relation of all part numbers with total cost and mass, in *Answer*. Even at this point, the SQL code starts to look considerably more forbidding than our previous Pascal solution.

It is not hard [Nikhil 84] to combine the operators of the relational algebra with some form of function definition in a simple interactive language to provide sufficient computational power to solve Task 3; and one wonders why this is not done more often. We shall also see that in section 3.7 that the relational algebra can be extended to provide a reasonably concise solution to this task. We mentioned in the introduction that two reasons are often given for the failure of most relational query languages to provide adequate power to solve Task 3: one is that "end users" will (for whatever reason) be unable to learn a more complicated language, the other is that a relational language should be limited to the operations that some database processor can perform efficiently. It would seem as that this is the sort of program that an end user might reasonably want to write, and that it should be possible to formulate it. Moreover, this is precisely the sort of task that a database machine *can* perform efficiently and the only reason for excluding it is that by allowing general recursive programs, one is allowing certain computations that *might* not be efficiently implemented.

INSERT INTO *Part* <1234, "Mast">
INSERT INTO *CompositePart* <1234, 20, 150>

INSERT INTO *Use* <1234, 912, 50>
INSERT INTO *Use* <1234, 603, 1>
...etc

Figure 2-9: TASK 4: Recording how a new part is composed

Performing the update, Task 4, in SQL exemplifies the need for transactions and integrity constraints. For example, we need to ensure that the parts data remains acyclic and that the condition that a part is either base or composite is maintained. Moreover, it is up to the user to invent an appropriate part number (the need for which is engendered by the relational model). Figure shows the interaction that might take place if SQL were to be used for this task. We should note that this update would probably be preceeded by a query that found the relevant part numbers given, say, their names. The fact that this whole process is so prone to constraint violation and transcription errors means that in any working environment, this update would be implemented using an embedded language if it were to be performed at all taskfrequently.

The purpose of this section was to introduce four tasks that we believe are characteristic of database programming and to show two approaches that failed for different reasons to meet the challenge: Pascal because of inadequate persistence and lack of the appropriate data types; SQL because of lack of computational power and,

arguably, lack of an adequate data model. Whether some combination of these approaches can overcome these obstacles is the subject of the next section.

3. Existing Database Programming Languages

We now turn to solutions to our problem set that actually work. We shall discuss several approaches. The first is the traditional solution of communicating with the database through a set of subroutines. Codasyl database management systems are almost always used in this fashion and much database programming is done by using Codasyl subroutines from languages such as Cobol, PL/I and Fortran. In fact, preprocessors or modified compilers have been developed for these languages, Cobol in particular, that provide a more consistent surface syntax for these subroutine calls. However, we do not want to call Cobol-Codasyl a database programming language because the Codasyl schema declaration, the Data Definition Language is not part of the "type" declaration of Cobol. The honour of being the first successful integrated database programming language must go to Pascal/R [Schmidt 77]. Pascal/R is Pascal with an added relational data type. It has been used fairly extensively, and is the second representative language presented in this section. We shall then examine another commonly used method of database programming, that of embedded relational languages. Finally, we shall look at a number of interactive languages that, because of their persistence or types have some relation to database programming.

3.1. The Codasyl approach: the database as external subroutines

Providing access to the database through a set of subroutines is the usual method of database programming for all pre-relational database management systems, and is again common in micro-computer based database management. We shall use Codasyl as an illustration since it prescribes one of the most sophisticated interfaces. Since the host languages have inadequate type systems for describing the database, this must be done in a separate language, the Data Definition Language (DDL). In Codasyl the DDL performs the task of physically describing the database layout and of describing the basic logical relationships. From this, subschemas, or views, may be generated through further use of a DDL. Most database text-books, e.g. [Date 81b] describe how this is done. In general the quality of languages against Codasyl databases has been somewhat worse than that against relational databases, their development having been retarded by the complexity of the details in the Codasyl specifications. A survey is to be found in [BCS 81].

The Codasyl interface is most commonly used from COBOL, PL/I or FORTRAN. Neither of these languages have parameter structures, functions with the appropriate types, nor adequate control structures to permit a good assimilation of the Codasyl

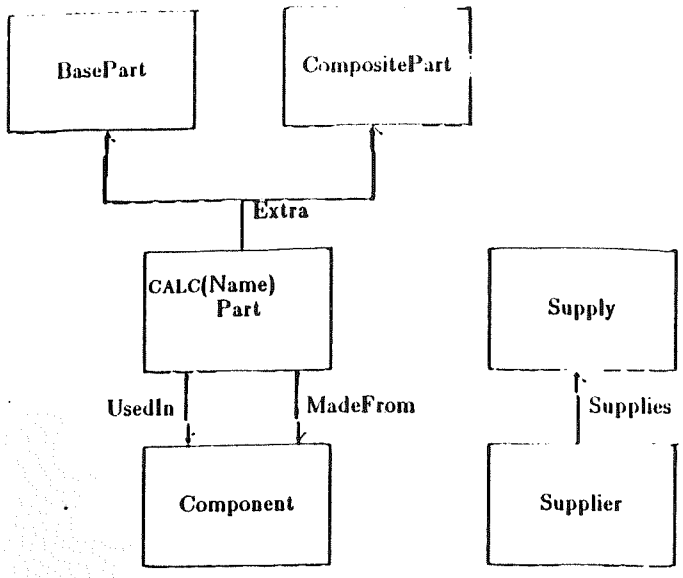

Figure 3-1: Task 1: Codasyl approach, Bachmann diagram

database model (even if it were simplified) into the language. We will not illustrate these existing interfaces, as this would consume space without shedding much light on types or persistence. The interested reader is refered to [Atkinson *et al.* 84] for an example of such an interface. However, we will show how an efficient and convenient interface can be constructed using the types of Pascal. To prepare for this: figure 3-1 shows the Bachmann diagram for our example database and figure 3-2 shows the corresponding schema[7]. Note that some of the referential constraints in figure 2-6 are implicit in Codasyl and others are made explicit with **MANDATORY** set membership. We are not aware of an implementation in which it is possible to express the constraint that a *Part* should not be both a *CompositePart* and a *BasePart*. There are two choices for representing the specialisation hierarchy (represented as a variant record in our Pascal data - see figure 2-1) in the Codasyl approach:

1. Place all the fields in one record to represent all parts, encoding *not applicable* with some appropriate null value - similar to an alternative we described for relations.

2. Use a set to relate the most general type to other records that contain the

[7] For those who need to know more about Codasyl DBMS, tutorial material and discussion can be found in [Olle 78]

```
SCHEMA name is PartsDB.
AREA  PartsAssembly.
AREA  PartSupply.

DATABASENAME PartName; PICTURE A(16).

RECORD type Part;                          {one for every part}
     WITHIN area PartsAssembly;
     LOCATION mode is CALC on Name USING PartName;
     DUPLICATES are NOT allowed;
     02 Name;    PICTURE A(16).

RECORD type BasePart;                      {one for each imported part}
     WITHIN area PartsAssembly; LOCATION is VIA Extra;
     02 UnitPrice;    PICTURE 9(4);
     02 Mass;         PICTURE 9(7).

RECORD type CompositePart;                 {one for each assembled part}
     WITHIN area PartsAssembly; LOCATION is VIA Extra;
     02 AssemblyCost;  PICTURE 9(5);
     02 MassIncrement; PICTURE 9(6).

RECORD type Use;                           {one for each use of a part}
     WITHIN area PartsAssembly; LOCATION is VIA UsedIn;
     02 Quantity;    PICTURE 9(4).

RECORD type Supply; LOCATION mode is VIA PMII;
     WITHIN area PartSupply;
     ...

RECORD type Supplier;
     LOCATION mode is CALC on ...

SET Extra; OWNER Part;                     {represent type hierarchy}
     MEMBER BasePart;
     MEMBERSHIP is MANDATORY AUTOMATIC;
     SELECTED by CALC USING PartName;
     MEMBER CompositePart;
     MEMBERSHIP is MANDATORY AUTOMATIC;
     SELECTED by CALC USING PartName.
SET MadeFrom; OWNER Part;                  {represent part explosion}
     MEMBER Use;
     MEMBERSHIP is MANDATORY AUTOMATIC;
     SELECTED by CURRENT of OWNER.
SET UsedIn; OWNER Part; MEMBER Use;
     MEMBERSHIP is MANDATORY AUTOMATIC;
     SELECTED by CALC USING PartName.
SET Obtains; OWNER CompositePart; MEMBER Supply;
     MEMBERSHIP is MANDATORY AUTOMATIC;
     SELECTED by CURRENT of OWNER.
SET Supplies; OWNER Supplier; MEMBER Supply;
     MEMBERSHIP is MANDATORY AUTOMATIC;
     SELECTED by CALC USING PartName.
```

Figure 3-2: Task 1: Codasyl DDL describing the parts data

additional information appropriate to its specialisation. Note this allows specialisation to be overlapping or exclusive.

The second option is taken in the example given here by using the set *Extra*. Since

reference, via sets, is supported, it is not been necessary to introduce *Pno*, an artificial key, as in the relational example.

Readers familiar with Bachmann diagrams will recognise the structure formed by the record types *Part* and *Component* and the two sets, *UsedIn* and *MadeFrom*. It ususally indicates the presence of some kind of graph structure (in this case an acyclic graph on the parts) and is quite common in the database designs we have observed. Its presence generally implies the kind of recursive processing we are using as our solutions to Task 3.

```
type ExtraClass      = record
                         case RecordType: (BasePartType, CompositePartType) of
                           BasePartType: (BasePartVal:BasePartRef);
                           CompositePartType: (CompositePartVal:CompositePartRef)
                       end;
type BasePartRef     = record
                         found: boolean;
                         ... {control information}
                       end;
type CompositePartRef= record
                         found: boolean;
                         ... {control information}
                       end;
type PartRef         = record
                         found: boolean;
                         ... {control information}
                       end;
type UseRef          = record
                         found: boolean;
                         ... {control information}
                       end;
type BasePart        = record
                         UnitPrice: 0..9999;
                         Mass: 0..99999999
                       end;
type CompositePart   = record
                         AssemblyCost: 0..99999;
                         MassIncrement: 0..9999999
                       end;
type Part            = record
                         Name: packed array[1..16] of Char
                       end;
type Component       = record
                         Quantity: 0..9999
                       end;
...
```

Figure 3-3: Pascal types automatically generated from a Codasyl Schema

The considerable investment in existing databases and in existing programming languages means that developing better techniques for their combined use remains important. The use of Pascal with Codasyl is illustrated here, using an interface described in [Buneman *et al.* 82a], which has a number of advantages. It exploits the

```
procedure GetPart(InRef: PartRef; var OutRec: Part);

procedure FindFirstInAreaPart(var OutRef: PartRef);

procedure FindNextInAreaPart(InRef: PartRef; var OutRef: PartRef);
... (similarly for other record classes and other Codasyl verbs.)

procedure FindByKeyPart(KeyVal: PartName; var OutRef: PartRef);

... (procedures for the MadeFrom set)
procedure FindFirstInSetMadeFrom(InRef: PartRef; var OutRef: UseRef);

procedure FindNextInSetMadeFrom(InRef: UseRef; var OutRef: UseRef);

procedure FindOwnerInSetMadeFrom(InRef: UseRef; var OutRef: PartRef);
... (etcetera for other Codasyl verbs and sets)

... (standard procedures for open, close for each area etc.)

... (standard procedures startTransaction, commitTransaction etc)
```

Figure 3-4: Pascal declarations automatically generated from the Codasyl Schema

strong static typing of Pascal programs so that the Pascal compiler itself prevents many of the errors common in programming with Codasyl databases. The technique is to generate *automatically* a set of named types and procedures from the Codasyl schema (or subschema). The form of the generated types can be seen in figure 3-3, which is part of the data description that would be generated from our sample schema (see figures 3-1 and 3-2). Note that the names of the types are constructed from the names in the schema in a systematic way. Record type names in Pascal are the same as the corresponding schema record type names. Field names within the record are the same as the names in the corresponding schema record, and as near as possible have an equivalent type to those in the original schema. An extra set of types, to be used as tokens equivalent to database currencies in Codasyl are introduced with the postfix *Ref*.

A sample of the corresponding procedures appears as figure 3-4. Note that the procedure name encodes the operation (first) and the operand (record type and area or set type) second. The second part is the name of the set or record type copied from the schema[8]. The types of the parameters ensure that the appropriate type of currency is supplied and the appropriate type of currency or record is yielded. A variant record has been automatically introduced to accomodate the set *Extra* which has more than one type of member. Thus our representation of specialisation automatically maps to Pascal's corresponding construct. This set of generated types and procedures has linked

[8] in the actual implementation there is some abbreviation

the data definition in the program in two ways: it has defined the mapping of types and data between database and programming language; and it has made the names introduced in the schema available in the programming language's name space, arranging that they can only be used for operations that are appropriate to their values. This relieves the programmer of the need to set up a mapping and therefore reduces the number of mistakes that can be made. However, the programmer still has to understand the Codasyl schema, which is quite different from the notation and structures familiar in Pascal.

The code for Tasks 2 and 3 has the same structure as that of the (pure) Pascal code given earlier. However pointer manipulation has been replaced by various calls to external subroutines. In Task 2 the variable *aBasePartRef* is a Pascal record that contains currency information needed by the Codasyl data manipulation subroutines and a field named *found* that indicates, after any *Find* operation, whether an appropriate database record was found. Thus the test *aBasePartRef.found* corresponds to the test $pt \neq nil$ in figure 2-2, and the call to *GetPart* corresponds to dereferencing in the same figure. Again, Task 3 exhibits the same complexity as the Pascal program. It is in this kind of task that other Codasyl interfaces look even more complicated because the programmer must explicitly control currency changes during recursion, and must take care not to confuse currencies of different "types", since they are usually all typed the same way, for example as integers.

The code to create and store a new composite part (figure 3-7) is probably more cumbersome than it would be in more traditional Codasyl interfaces. In these interfaces the procedures to store records are essentially unparameterised, and operate by inspecting and modifying a global state. For this example, in contrast to Task 3, the side effects created by one database operation would produce the correct global state for the next operation, thus some of the variables figure 3-6 would be not be needed. In the solution presented here, the first parameter of any *Store* ... procedure is the record to be stored, the last is a data base reference which is set to refer to the record once it is stored. The intermediate parameters are references to owner records, one for each set in which the record being stored is a member. Sets are referenced by position. This is slightly unsatisfactory, but is probably better than constructing a record type in which the sets could be named. A better solution is to use keyword parameters, and an Ada interface for Codasyl, also specified in [Buneman *et al.* 82b], is considerably less cumbersome because one can use keywords and exploit overloading so that, for example,

```
program task2;

...{Type declarations from figure 3-3}

...{Declarations of external procedures listed in 3-4}

var aBasePartRef: BasePartRef; aBasePart: BasePart;
    aPartRef: PartRef; aPart: Part;

FindFirstInAreaBasePart(aBasePartRef);
while aBasePartRef.found do
  begin
    GetBasePart(aBasePartRef, aBasePart)
  ; if aBasePart.UnitPrice > 100 then
      begin
        FindOwnerInSetExtra(aBasePartRef, aPartRef)
      ; GetPart(aPartRef, aPart)
      ; writeln( aPart.Name, aBasePart.UnitPrice, aBasePart.Mass)
      end {if}
  ; FindNextInAreaBasePart(aBasePartRef, aBasePartRef)
  end; {while loop}
end.
```

Figure 3-5: Task 2: Listing all the expensive parts using the Pascal interface to Codasyl

all the store procedures are called *Store*.

3.2. Pascal/R: a true Database Programming Language

This section uses Pascal/R to illustrate a group of languages often called *integrate* DBPLs [Pirotte 80, Atkinson *et al.* 84], which attempt to combine a general-purpos programming language with a data model[9] as consistently as possible. The level o integration depends on the degree to which the designers and implementers wer prepared to change the language they started with (most chose to start with Pascal.) A such languages are fairly numerous, the attempts to integrate programming language with the relational model are summarised in figure 3-8.

Pascal/R has been fairly widely used in universities for research and teaching, and has also been used for some commercial applications. It is also the starting point for Modula/R [Koch *et al.* 83] and for the present work of Schmidt's group, DBPL [Schmidt & Mall 83], and so it is chosen as the vehicle for our examples in this category.

The first problem facing the integrated language designer, is how to relate the two type systems: that of the data model, and that of the PL. In Pascal/R the tuple is identified

[9] Rather than with an existing, external, database as in sections 3.1 and 3.4


```
program task3;

...{Type declarations from figure 3-3}

...{Declarations of external procedures listed in 3-4}


procedure CostAndMass(p: PartRef; var resultCost, resultMass: integer);
var cRef: ExtraClassRef; cRec: ExtraClass;
    bRec: BasePart; cRec: CompositePart;
    uRef: UseRef; uRec: Use;
    pRef: PartRef;
    subTotalCost, subTotalMass: integer;

begin
  FindFirstInSetExtra(p, cRef);
  Get(cRef, cRec);
  case crec.RecordType of
    BasePartType:
      begin
        GetBasePart(cRec.BasePartVal, bRec);
        resultCost := bRec.UnitCost; resultMass := bRec.Mass
      end;                                  {of base part case}
    CompositePartType:
      begin
        GetCompositePart(cRec.CompositePartVal, cRec);
        resultCost := cRec.AssemblyCost; resultMass := cRec.MassIncrement;
        FindFirstInSetMadeFrom(cRec.BasePartVal, uRef);
        while uRef.found do
          begin
            GetComponent(uRef, uRec);
            FindOwnerUsedIn(uRef, pRef);
            CostAndMass(pRef, subTotalCost, subTotalMass);
            resultCost := resultCost + CompRec.Quantity*subTotalCost;
            resultMass := resultMass + CompRec.Quantity*subTotalMass;
            FindNextInSetMadeFrom(uRef, uRef)
          end{while more subcomponents}
      end{dealing with CompositePartType}
  end {case of either specialisation}
end; {procedure CostAndMass}

procedure main;
var thePart: PartRef; itsCost, itsMass: integer;
begin
  FindByKeyPart('Mast       ', thePart);
  if not thePart.found then writeln('Couldn''t find it')
  else
    begin
      CostAndMass(thePart, itsCost, itsMass);
      writeln(itsCost, itsMass)
    end
end; {main}

begin main end.
```

Figure 3-6: Pascal program to compute mass and cost as in Task 3

with the record and **relation of** is introduced as a constructor similar to **set of**. It takes two parameters, the type of the record which may be a tuple of the relation, and the subset of that record's fields that constitute the primary key (there are restrictions on

```
program Task4;

...{Type declarations from figure 3-3}

...{Declarations of external procedures listed in 3-e}

procedure doTask4;
type inputListType = ↑InputCell;
     inputCell = record
                   Name: packed array [1..16] of Char;
                   Quantity: 0..9999;
                   Next: inputListType
                 end;

var inputList: inputListType;
    newName: packed array [1..16] of Char;
    newAssemblyCost: 0..99999;
    newMassIncrement: 0..9999999;
    {inputList is a list of names of the subparts of the new part
       together with their quantities}

    aPart: Part; aPartRef, aSubPartRef: PartRef;
    anExtra: ExtraClass; anExtraRef: ExtraCLassRef;
    aCompositePart: CompositePart; aCompositeRef: CompositePartRef;
    aComponent: Component; aComponentRef: ComponentRef;
begin

...{code to interact with the user and get values for newName,
    newAssemblycost, newMassIncrement and inputList}

  with aPart do Name:=newName;    {Create and store the new part}
  StorePart(aPart, aPartRef);
  with anExtra do begin           {Create and store the variant}
    RecordType:=CompositePartType; CompositePartVal:=aPartRef
  end;
  StoreExtraClass(anExtra, anExtraRef);
  with aCompositePart do begin
    MassIncrement:=newMassIncrement; AssemblyCost:=newAssemblyCost
  end;
  StoreCompositePart(aCompositePart, anExtraRef, aCompositeRef);

  while inputList ≠ nil do begin {Now establish links to sub-parts}
    findByKeyPart(inputList↑.Name, aSubPartref);
    aComponent.Quantity:=inputList↑.Quantity;
    inputList:=inputList↑.Next;
    StoreComponent(aComponent, aPartRef, aSubPartRef, aComponentRef);
  end
end;

begin doTask4 end.
```

Figure 3-7: Pascal program to update a Codasyl database

the types allowed as such a record's fields - we discuss the impact of the restriction later).

Another problem to be overcome, is how to identify databases, and how to introduce names from the database into a program. Pascal/R added the **database** constructor to

System	Starting PL	DML	status
Pascal/R [Schmidt 77]	Pascal	Calculus	working prototype widely used
Aldat/MRDS [Merrett 84]	[illegible]al alg.	Algebra	working prototype
Astral [Amble *et al.* 79]	Pascal	Algebra	proposal
Rigel [Rowe & Shoens 79]	Pascal	Calculus	partially implemented
Theseus [Shopiro 79]	Euclid [Lampson 77]	Calculus	proposal
Plain [Wasserman *et al.* 81]	Pascal extended [Wasserman 81]	Algebra	partially implemented
Modula/R [Koch *et al.* 83]	Modula-2 [Wirth 83]	Calculus	product in wide use
AdaRel [Horowitz & Kemper 83]	Ada [Ichbiah *et al.* 79]	?	Proposed

Figure 3-8: Combining Existing Languages with the Relational Model

Pascal. This constructor is parameterised in a manner similar to the record constructor except that all fields have to be of type relation (another restriction considered later). A database variable may then be declared in the program, and appear in the program's parameters (c.f. **file** parameters in Pascal). This allows the same program to be run with different databases *of the same type*, and introduces the relation names for that database into the program. The necessary type description of those relations ensures that their column names are available in the program.[10]

The designer has also to decide what manipulations to provide. In Pascal/R relations may be arguments of procedures, and there are relational operators that correspond to relational calculus. The transition between these bulk operations (selected because of their potential for efficient implementation [Jarke & Koch 82, Jarke & Koch 83, Bragger *et al.* 83, Jarke & Koch 84]) and the iterative control structures of Pascal is made using the new iteration construct **for each**. Unfortunately this construct is not consistently available for the other repetitive structures in Pascal/R, such as **files** and **arrays**.

[10] The implementation restricts a program to operating on only one DB, and prevents that from being chosen dynamically - two fairly inconvenient restrictions.

Another mechanism based on four procedures: *low*, *next*, *high*, and *eor* allows for explicit control over the traversal of a relation.

```
type                                          {domains}
  Pnum   = 1 .. maxint,                       {Part numbers}
  String = packed array [1..16] of char;
  Dollars = real;
  Grams  = real;
  ...
  {records destined to become tuple types}

  PartRec          = record                   {one per part, base or composite}
                       Pno: Pnum;
                       Name: String
                     end;
  BasePartRec      = record                   {one per bought in part}
                       Pno: Pnum;             {in PartRec}
                       UnitPrice: Dollars;
                       Mass:Grams
                     end;
  CompositePartRec = record                   {one per constructed part}
                       Pno: Pnum;             {in PartRec, not in BasePartRec}
                       AssemblyCost: Dollars;
                       MassIncrement: Grams
                     end;
  MadeFromRec      = record                   {one per Part/SubPart relationship}
                       Assembly: Pnum; {in CompositePart}
                       Component: Pnum; {in Part}
                       Quantity: PosInt
                     end;
  SuppliedBy       = record                   {one for each supplier of each Part}
                       Pno: Pnum;             {in BasePart}
                       Sno: Snum              {in Suppliers}
                     end;
  ... {suppliers etc}
      {relation types}
  PartRel          = relation <Pno> of PartRec;
  BasePartRel      = relation <Pno> of BasePartRec;
  CompositePartRel = relation <Pno> of CompositePartRec;
  MadeFromRel      = relation <Assembly, Component> of MadeFromRec;
  SuppliedByRel    = relation <Pno, Sno> of SuppliedByRec;
  ...
      {The data base type}
  PartsDB  =                                  database
             Part: PartRel;                   {known parts}
               BasePart: BasePartRel;         {parts bought in}
               CompositePart: CompositePartRel;{complementary set ofconstructed parts}
               MadeFrom: MadeFromRel;         {parts to make a part}
               SuppliedBy: SuppliedByRel;     {who can supply each base part}
               ...
           end;
```

Figure 3-9: Task 1: describing the data in Pascal/R

Figure 3-9 shows the data definition corresponding to Task 1. Note the similarity with the relational description in figure 2-5. This similarity extends to the inability to describe the referential and inheritance semantics shown in figure 2-6. However, the notation here allows the component types to be named. This is not necessary, but it is

advantageous as it introduces names for types which are needed for programming with this data. Note the difference from the Pascal example in figure 2-1 where variants are used, capturing the semantics of the required and exclusive specialisations of *Part*. The version of Pascal/R available did not adhere to data type completeness and allow a relation to be constructed over variant records, presumably because it is then more difficult to define the operations on relations in all circumstances. The best that could be done was to add a few comments.

```
program Task2(aPartDB);
  ...                                         {the type decls from fig 3-9}
  var aPartDB: PartsDB;
  ...
  with aPartDB do                             {relation names into scope}
    for each bPart in BasePart: UnitPrice > 100 do
      writeln( bPart.UnitPrice, bPart.Mass )
```

Figure 3-10: Pascal/R - partial solution to Task 2

The program in figure 3-10 shows how to obtain the price and mass of each base part. This is relatively simple. However, as there is no inheritance mechanism, an explicit join is necessary to find the name, as shown in figure 3-11.

```
program Task2(aPartDB);
  ...                                         {the type decls from fig 3-9}
  var aPartDB: PartsDB;
  ...
  with aPartDB do                             {relation names into scope}
    for each bPart in BasePart: UnitPrice > 100 do
      for each prt in Part: bPart.Pno = prt.Pno do
        writeln( prt.Name, bPart.UnitPrice, bPart.Mass )
```

Figure 3-11: Pascal/R - Task 2 printing name of expensive parts

Note that the types of the controlled variables in the **for each** statement are *bPart:BasePartRec* and *prt:PartRec*. The automatic typing of these variables is convenient, and their use avoids ambiguity in the selection clause and in the iterated statement. However, these intermediate type names are useful, as is shown in the example of a function to locate a part, shown in figure 3-12. Both the relation and the record type are used. That example also shows that obtaining a singleton set, then *deselting* to obtain the one item it holds (a common operation in the applications tried), is not particularly convenient in Pascal/R. Later, we will see a construct, **the**, in Daplex and Adaplex, which has this purpose.

The Task 3 computation of cost and mass is shown in figure 3-13. There is the equivalent of a join in this computation, but it is not written as such. The programmer

```
program Task3(aPartDB);                                  {type decls from fig. 3-9}
 ...
 var aPartDB: PartsDB;
 function findPart(partName: String): Pnum; {returns part number}
 var aPart: PartRec; theParts: PartRel;
 with aPartDB do
                                                         {relation names now available}
  begin
   theParts := [each prt in Part: prt.Name = partName];
   if size(theParts) ≠ 1 then                            {error}
    begin
     writeln ('Ambiguous or unknown part name');
     findPart := MaxInt                                  {representing unknown}
    end                                                  {of error}
   else
    begin
     low( theParts, aPart);                              {desetting}
     findPart := aPart.Pno
    end
  end
end;                                                     {of findPart}
```

Figure 3-12: Pascal/R - function to locate a part

```
program Task3(aPartDB);                                  {type decls from fig. 3-9}
 ...
 var aPartDB: PartsDB;
 ...                                                     {findPart from fig 3-12}
 procedure costAndMass(p: Pnum; var resultCost: Dollars;
                       var resultMass: Grams);
  var subTotalCost: Dollars; subTotalMass: Grams; mf: MadeFromRec;
  with aPartDB do
   begin
    if BasePart[p] in BasePart then                      {is it a base part?}
     begin                                               {base part case}
      resultCost := BasePart[p].UnitPrice;
      resultMass := BasePart[p].Mass
     end                                                 {of Base Part case}
    else
     begin                                               {here a composite part}
      resultCost := CompositePart[p].AssemblyCost;
      resultMass := CompositePart[p].MassIncrement;
      for each mf in [MadeFrom: mf.Assembly = p]  do
       with mf do
        begin
         costAndMass( Component, subTotalCost, subTotalMass );
         resultCost := resultCost + Quantity * subTotalCost;
         resultMass := resultMass + Quantity * subTotalMass
        end                                              {of loop over components}
     end                                                 {of Composite Part case}
   end                                                   {of scope of aPartDB relation names}
  end;                                                   {of costAndMass}

 procedure doTask3;
  var itsCost: Dollars; itsMass: Grams;
  begin
   costAndMass( findPart( 'Mast '), itsCost, itsMass);
   writeln (itsCost, itsMass)
  end;                                                   {of doTask3}

begin doTask3 end.
```

Figure 3-13: Pascal/R - program to obtain cost and mass

has kept track of one of the contributing tuples, to use its values directly. This also

saved the programmer defining the type that would result from the join. It is difficult to provide a single operator that will allow this addition, multiplication and recursion. We will see a partial solution in section 3.7. The operation *CompositePart[p]* used in this example, treats the relation as a sparse array, rather than as a set. This can also be seen as treating it as an extensionally defined function. A useful idea we will see again in Daplex.

```
program Task3(aPartDB);                                  {type decls from fig. 3-9}
 ...                                                     {to hold previously computed cost etc}
 memoRec = record                                        {in CompositePart}
             ForPart: Pnum;
             TotCost: Dollars;
             TotMass: Grams
           end;
 memoRel = relation <ForPart> of memoRec;
 var aPartDB: PartsDB; memo: memoRel;
 ...                                                     {findPart from fig 3-12}
 procedure costAndMass(p: Pnum; var resultCost: Dollars;
                       var resultMass: Grams);
  var subTotalCost: Dollars; subTotalMass: Grams; mf: MadeFromRec;
  with aPartDB do
   begin
    if BasePart[p] in BasePart then                      {is it a base part?}
     ...                                                 {as for fig 3-13}
    else
     if memo[p] in memo then                             {memoised?}
      with memo[p] do
       begin                                             {here if already calculated}
        resultCost := TotCost;
        resultMass := TotMass
       end
     else                                                {not previously computed}
      begin                                              {here a composite part}
      resultCost := CompositePart[p].UnitPrice;
      resultMass := CompositePart[p].Mass;
      for each ...                                       {as for fig 3-13}
        ...                                              {as for fig 3-13}
                                                         {memoise values for this part}
      memo :+ [<p, resultCost, resultMass>]
     end                                                 {of Composite Part case}
   end                                                   {of scope of aPartDB relation names}
  end;                                                   {of costAndMass}
 procedure Task3;
  var ...
  memo := [];                                            {nothing calculated yet}
 ...                                                     {as for fig 3-13}

end
```

Figure 3-14: Pascal/R - memoising the cost and mass

To avoid the repetitive computation for common substructures[11], a memo data

[11] good engineering practice tries to reuse common subassemblies, so this saving can be significant

structure[12] can be introduced and used, as shown in figure 3-14. Note that it is easy t declare and initialise the necessary set of values. The :+ operator is used to add a new result to *memo*, each time a composite part is calculated. The construct *memo*[*p*]) in *memo* to determine whether the part has already been calculated is not obvious as a construct for determining whether the index value is in the set. It was also used to determine whether a part was a composite part. It depends on *memo*[*p*] returning the empty tuple <>, if the index is not present, and this tuple is not considered to be present in any relation.[13] It is convenient to be able to declare in the program the *memo* relation of the type initially specific to the database, but to have the instance extant only for each program run. All the same mechanisms are available for dealing with temporary relations, such as *memo*, as are available for persistent relations.

```
type
    Pnum     ...                          {domains as in fig 3-9}
    PartRec  ...                          {records as in fig 3-9}
    IdRec    = record                     {to package a Pnum}
                 Nom: (NextPno, NextSno, ... );
                 Value: integer
               end;
    PartRel  ...                          {relations as in fig 3-9}
    IdRel    = relation <Nom> of IdRec;
    PartsDB  = database
                 Part: ...                {as in fig 3-9}
                 ...
                 Id: IdRel;               {next identifier to use}
                 ...
               end;
```

Figure 3-15: Pascal/R - revising data description for update

In order to perform the update task, it is necessary first to modify the database definition of figure 3-9 to include a record of the part numbers used. This is shown in figure 3-15 where the required integers are encapsulated in a dummy relation *Id*. It is unfortunate that type independence was not maintained to the extent that we could write

```
PartsDb =  database
           Part: ...
           ...
           NextPno: Pnum;      {the next part number to use}
           NextSno: Snum
           etc
           end;
```

directly recording the integers in the database. The update program implementing Task

[12] A mechanism originally suggested by Michie [Michie 68]

[13] It is interesting to ask what the type of the empty tuple is since In presumably has type $T \times$ relation of $T \rightarrow$ bool.

4 is shown in figure 3-16.

```
program Task 4(aPartDB);
 ...                                  {type def[n] as in fig 3-15}
 ...                                  {the function findPart as in fig 3-13}
var aPartDB: PartsDB;
procedure makeSub( assemblyPno, componentPno: Pnum; q: PosInt );
begin
  with aPartDB do
    begin
      if not Part[ComponentPno] in Part then
        ... {Stop the transaction with an "Unknown component" message}
      MadeFrom :+ [<assemblyPno, ComponentPno, q>]
    end
end;                                   {of makeSub}

procedure Main(newName: String; newAssemblyCost: Dollars;
                 newMassIncrement: Grams);
  var newPart: PartRec; newCompositePart: CompositePartRec;
      cmpntList: relation <cPno> of
                   record cPno: Pnum; Qty: Posint end;
  begin
    ...{code to obtain from the user the details of a part,
       the composite part information, and the list of components
       used in its manufacture together with quantities}
    with aPartDB do
      begin
        with Id[ NextPno ] do
          begin
            newPart.Pno:=value;            {see fig 3-15}
            Part :+ [newPart];
            newCompositePart.Pno := value;
            CompositePart :+ [newCompositePart];
            for each mf in cmpntList do
              makeSub(value, mf.cPno, mf.qty )
          end                              {of with Id[ NextPno ]}
      end                                  {of with aPartDB}
  end;                                     {of Main}

begin Main end.
```

Figure 3-16: Pascal/R - Task 4 recording how a part is made

This code is not typical of an update program, which would be largely concerned with providing a suitable user interface, and checking that the user's input made sense. For example, there is no check here that cycles are not being introduced into the intended directed acylcic. If concurrent use of the database were anticipated, it would be important not to lock too much while the user did input. However, that does not arise with Pascal/R as there is no indication of the extent of a transaction. The implementation actually treats a whole program run as a transaction. One deficiency of Pascal/R not exhibited in the examples, is the constraint that only one database may be used in any given program. This seems serious, as people often require to combine data from different databases, produce copies etc. The main impact would be on the

introduction of names. Since Pascal/R treats databases like records, there would be no problem of disambiguating names, and so we assume that a change to permit multiple database use would not be difficult.

3.3. Other languages that attempted integration with relations

Many other languages have been proposed to combine a standard language with relations. A few comments are offered on some of those that appear in table 3-8.

In the late 70s, there was considerable progress reported with the languages Plain and Rigel. Plain was a particularly ambitious project, tackling a number of issues recognised as relevant to making the implementation of interactive information systems easier. The relations are similar to those in Pascal/R, with similar restrictions as to the attribute types. One extension is to introduce a *marking*, which refers to a subset of some base relation(s) derived by evaluating expressions in the relational algebra over relations and markings. Associative access on the key fields is supported in a fashion equivalent to that in Pascal/R. The provision of exceptions should in principal, make handling relation access errors easier than in Pascal/R. But standard exceptions for relational operations are not defined in the report [Wasserman *et al.* 81]. The **foreach** statement is consistently defined to be applicable to all composite objects, aggregating sets or sequences of values of the same type, rather than just over relations as in Pascal/R. In addition there is a limited way of specifying the order of iteration, similar to that we will see later in Adaplex4.1. Relational update operators almost identical with those in Pascal/R are available. Persistence greater than the duration of a program is achieved by binding a declaration to an *external* object in the environment. It appears that this was only intended for a limited subset of types, such as procedures, modules and relations. The authors understand that the underlying database handler intended for Plain has been widely used [Kersten 81].

Using the relational calculus rather than algebra, Rigel was built on top of INGRES [Stonebraker *et al.* 76]. Its type system is reported in [Rowe & Shoens 79]. Interest moved from the database facilities, to means of facilitating programming interaction against relations via forms [Rowe 85].

The languages Theseus [Shopiro 79] and Astral [Amble *et al.* 79] were designed, but we understand their implementation was not completed.

Various plans for adding database facilities to Ada have been mooted, besides Adaplex, reported later. The suggestion for Ada and relations, Adarel [Horowitz & Kemper 83], is so recent that an implementation is unlikely to be finished yet. It is not known whether the proposal for a persistent Ada [Hall 83] , with a similar notion of persistence to PS-algol(see Section 6.1, is being implemented, but an interest in a similar idea is reported at CCA, implementers of Adaplex.

3.4. Embedded Languages

This section, describing existing methods of database programming, would not be complete without some reference to embedded languages. Confronted with the failure of a query language to handle a problem such as Task 3, or with the difficulty of performing a safe update, the user of a relational database would normally turn to an embedding of the query language in some more powerful language such as Cobol, PL/1 [Date 83a], or C [Stonebraker *et al.* 76]. In such systems, queries can be interspersed with a host language in such a way that makes them recogniseable by a precompiler that generates the appropriate external subroutine calls. Since we have been using Pascal as our reference language, the solution to Task 3 is shown in figure 3-17 using a hypothetical embedding of SQL in Pascal. It follows closely the code that would be used for the embedding of SQL in PL/1 in System-R [Astrahan 76]. In this example, we have only shown the code for the procedure *CostAndMass*. The code to drive this procedure would also have to be written in the host language.

In this example a special class of variables, prefixed with a $, serves as communication between the host and query language. Since the host language has no data type appropriate for representing the result of a query when this is a relation with more than one tuple, a cursor (*X*) is used to traverse the relation in a fashion similar to reading a file.[14]

The code for this task combines two languages that we have already seen, and while logically straightforward is awkward because the programmer must be bilingual. The main problem with embedded languages is that data may only be conveyed across the interface via variables of a very limited set of data types such as integer, character string, and real. To our knowledge, no attempt has been made to establish a higher-level

[14]Note that the cursor is not an object in the host language and therefore cannot be declared with an appropriate type.

```
procedure CostAndMass(P: Pnum; var resultCost, resultMass: integer);
var $Pno, $Cost, $Mass, $Component, $Quantity,
    $subTotalCost, $subTotalMass: integer;
begin
 $Pno:=P;
 $SELECT UnitPrice, Mass
 INTO $Cost, $Mass
 FROM BasePart
 WHERE Pno = $Pno;
 if ERRORSTATUS = 0 then          {we found a base part}
  begin
    resultCost:=$Cost;
    resultMass:=$Mass;
  end
 else
  begin                           {we assume we have a composite part}
   $SELECT AssemblyCost, MassIncrement
   INTO $Cost, $Mass
   FROM CompositePart
   WHERE Pno = $Pno;
   resultCost:= $Cost; resultMass:=$Mass;
   $LET X BE                      {Define a new relation of subparts}
    SELECT Quantity, Component
    INTO $Quantity, $Component
    FROM Use
    WHERE Assembly = $Assembly;
   $OPEN X;                       {Set cursor to start of relation}
   while ERRORSTATUS = 0 do
    begin
     $FETCH X;
     CostAndMass($Component, subTotalCost, subTotalMass)
     resultCost:=$Quantity*subTotalCost+resultCost;
     resultMass:=$Quantity*subTotalMass+resultMass;
    end
  end
end;
```

Figure 3-17: Task 3 in a hypothetical embedded query language

interface by establishing, say, a tuple to record or relation to array correspondence.

3.5. Logic Programming

There is a well-established and elegant connection between logic programming and relational databases [Gallaire *et al* 84, Gallaire & Minker 78]. In particular, one can represent a relation as a set of simple predicates, all of the same arity, whose variables are all constants. Queries can be simply formulated and there is no problem in writing a logic program to perform transitive closure. In fact the following program computes the

transitive closure of the part - sub-part relationship:[15] the development of logic programming with inheritance.

```
AnySub(x,y):-MadeFrom(x,y,_)
AnySub(x,z):-MadeFrom(x,y,_),AnySub(y,z)
```

Unfortunately the introduction of arithmetic that is required for Task 3 considerably complicates the program and requires the addition of "cuts" to control the evaluation. A problem very similar to Task 3 is worked out in some detail in [Clocksin & Mellish 81]

The connection between database constraints (such as functional dependencies, multi-valued dependencies etc.) and types is a subject that is greatly in need of research. It is again possible to model database constraints very elegantly in logic programming and to devise systems that do not allow the addition of a predicate -- a form of update -- should it violate these constraints. However, there is nothing corresponding to *static* type-checking available for logic programming, although proposals for such a type-checker have been given by Mycroft [Mycroft 84]. We also understand [Vassiliou *et al* 83, Ullman 85] that there are attempts to produce an effective interface between Prolog and a relational database management system in order to obtain, where possible, the efficiency provided by the latter. However, we are unaware how these systems will deal with persistence and modularity.

Thus, while logic programming clearly deserves consideration as a database programming language, the issues addressed in this work are orthogonal to the issues of types and persistence that are the focus of this paper. Current implementations of logic programming languages deal with persistence in very much the same fashion as Lisp implementations. However, we expect that this situation will change in the near future and progress with implementations of logic programming systems that can be combined with database management systems should be carefully watched.

[15] The problem of specifying parameters positionally can be acute in real applications, where relations tend to get extremely "fat", i.e. they contain a large number of columns. An interesting development is recent work [Ait-Kaci & Nasr 85] in which inheritance can be used in logic programming. Among other things, this permits the use of a keyword notation which combines more naturally with conventional relational query languages.

3.6. Persistence and Workspaces

Many of the languages we discuss in this paper - the database query languages in particular - are designed to be used *interactively*. In practice this means that an environment is provided which the user, in the normal course of work, never leaves. Editing, compiling/interpreting, debugging etc. are all performed within the language environment. Although it can be claimed that there is no difference in principle between this and the interactive interface provided by many operating systems for the traditional edit-compile-run cycle used for many programming languages, there are qualitative differences that are important for our examination of database languages especially with respect to persistence.

Particularly important in the design of interactive languages and their environments is that many of the services normally provided by the operating sytem, such as editors, links to files, etc. are available in the language environment itself and should be callable from the language. This means that some attempt to deal with peristence must be made within the language or language environment. In this context the programming environments for Lisp [McCarthy 62, Teitelman 75], Prolog and APL [Iverson 70] are interesting.

The simplest form of persistence, provided by all useful interpreters for these languages is some form of checkpoint-restart instructions. By using these a programmer may save the current state of the environment and resume it later. This is a simple form of all-or-nothing persistence, a more sophisticated version of which is provided by Poly (see section 5.2). Lisp is more interesting because certain objects such as functions (SEXPRs) have a textual representation. These can be saved in files and read in again in some subsequent session. By grouping collections of such objects in files a degree of modularity can be obtained. However the list structues that one normally associates with databases cannot easily be "flattened" into a textual representation. Therefore special purpose algorithms need to be developed for certain classes of structure. This can be done, but there is a consequent loss of generality of the form of persistence that is available.

APL deserves special mention both because of its type-checking and its persistence. Although it was designed primarily as a scientific programming language, it has been widely used as a programming language for the implementation of small and medium scale databases. One of the reasons for this is that it was one of the first sophisticated

interactive languages to become available on the brands of hardware on which databases are commonly needed. Another is that its workspace mechanism made it particularly easy to partition and share data.

In APL, the only non-primitive data types are functions and arrays. There are no reference types. Every data structure is therefore flat, and a workspace is no more than a collection of such structures together with their names. The penalty for this organization is that global references within functions cannot be bound statically and the language exhibits the usual anomalies of dynamic binding. Type checking in APL is also interesting because the only "bulk" data type - arrays - is uniformly typed. Thus the type of A and B in $A + B$ is

(array of *real*) or (array of *integer*)

rather than

array of (*real* or *integer*)

Type-checking (and optimization) need only be done once for addition of arrays. This is in contrast to Lisp where there is no mechanism for imposing a uniform type on the elements of lists. APL programs that are written to exploit arrays and operators will therefore have a very small overhead for type-checking.

The flatness of APL structures is similar to the first normal form requirement for relational databases. Given this, it is surprising that non-one has yet attempted to produce a relational database programming environment based on that of APL. One reason is that efficient implementation of relational operators requires an elaborate overhead of non-flat data structures, and these may be difficult or expensive to disentangle when copying objects from one workspace to another. It is possible that Aldat, which we examine in the following section, will be implemented along these lines.

3.7. Aldat

Aldat [Merrett 77, Merrett 84, Merrett 85b] is an attempt to build a language that is predominantly relational algebra, with minimal extensions to give more programming power. This is a contrast to all the other examples which are basically general purpose programming languages that have tried to develop persistent storage.

In contrast to these attempts to develop persistent storage for general-purpose programming languages is the possibility of extending a database query language to provide greater computational power. Most query languages are more than bare

interpreters of relational calculus or algebra, but they are nevertheless limited as we have seen in our efforts to solve Task 3. Aldat [Merrett 77] however, uses the relational algebra as the basis for an extended and generalized set of relational operators that, together with recursion and function definition, provide an interesting medium in which to apply the relational algebra to a variety of programming problems.

An example of Aldat's extensions to the operations of the relational algebra is a form of relational composition $R\ [X_1, \ldots, X_n$ **lcomp** $Y_1, \ldots, Y_n]\ S$ where $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_n$ are lists of column names of relations R and S respectively. The meaning of this is to take the cartesian product of relations R and S, select those tuples t for which $t[X_i] = t[Y_i]$, and then remove, by projection, the columns $X_1, \ldots, X_n, Y_1, \ldots, Y_n$. Given a binary relation, $R(a,b)$*, one may define

```
R2 <- R ujoin (R [b lcomp a] R)
R3 <- R ujoin (R2 [b lcomp a] R)
RClose <- R ujoin (R [b lcompa] RClose)
```

where ujoin means natural join. The first two of these definitions perform closures of depths two and three; the last is an allowable recursive form and is a definition of the transitive closure of R.

This expression of transitive closure is not powerful enough to solve Task 3 where we need to perform arithmetic in the course of forming the closure. To do this, in a fashion reminiscent of APL there is a generalization of the transitive closure operation that computes an "outer product" along the edges of a directed acyclic graph. The first line of 3-18 therefore finds the total number of any component involved in an assembly (be it a sub-part, sub-sub-part etc.). The next two lines creates a uniform relation with costs amd masses for all parts in the data base ("<-" is assignment, "<-+" is incremental assignment). The final line performs the appropriate join and reduction to give us the total cost and mass for each part.

```
Quantities <- closure Use(Assembly, Component, * of Qty meet +);

AllCandM [Pno,C,M <- Pno,UnitPrice,Mass] Base;
AllCandM [Pno,C,M <-+ Pno,AssemblyCost,MassIncrement] Base;

let TotalCost be equiv + of C * Qty by Component;
let TotalMass be equiv + of M * Qty by Component;

CostAndMass <- Pnum, TotalCost, TotalMass in
        (Quantities [Component lcomp Pno] AllCandM);
```

Figure 3-18: A Computation for Task 3 in Aldat

This method is something of a departure from other methods we have seen for performing Task 3. In the first place, the existence of a transitive closure operation should allow for optimisation which would certainly include a form of "memoising". This, and other aspects of transitive closure, are extensively discussed in [Merrett 84]. On the other hand, the computation produces *all* the costs and masses, which may be more than was required and which, in a shallow parts explosion graph, means that much more computation has been done than needed. It is interesting to note that we cannot directly extend the **closure** operation to use user-defined functions that will directly compute cost and mass.[16]

In addition to a transitive closure operator, Aldat contains iterators and allows, as we have seen, certain forms of function definition. Merrett [Merrett 85b] shows how areas as diverse as geometric computation and inferencing can be represented using an extended relational algebra. Aldat undoubtedly exhibits some form of computational completeness, and it would be interesting to have some theoretical characterisation of this.

[16] There is again an interesting similarity with APL. Higher level operations like reduction, outer product, inner product etc. need to know the *identity* of their function arguments (0 for +, 1 for * etc.) It is not obvious how the identity is to be specified for user defined functions.

4. Languages incorporating advanced data models

To this point we have looked only at the relational and network data models. The failure of these models to capture adequate database semantics is well understood [Codd 79] and other models, notably the entity-relation model [Chen 76], have been widely used as design tools for databases. Surprisingly, this data model does not appear to have had a very direct effect on the design of database programming languages. There are, however, two data relationships that have been found essential in database design and have been captured within the type system of several experimental database programming languages.

The notion of inheritance has received attention in various relevant fields and goes by various names: in Artificial Intelligence, *ISA* hierarchies have been expressed in semantic networks such as KL-One [Brachman 78, Brachman 83]; type (or class) *inheritance* is an important feature of object-oriented languages such as Simula [Dahl & Nygaard 66] and Smalltalk [Goldstein 80]; and *generalisation* (or *specialisation*) were suggested for databases by [Smith 77] and used as the basis for the Semantic Data Model [Hammer 75]. In our test case, an example of inheritance is to be seen in the relationship between *BasePart* and *Part*. A *BasePart* is a special kind of *Part* and therefore inherits all the attributes of *Part*. In the relational model, the only way to capture this relationship is by a "foreign key" constraint, which is not usually available in relational schema declarations. We used a variant record in Pascal to express this relationship, but this is in general too restrictive since it constrains (a) a part to be a base part or a composite part and (b) no part to be both base and composite. In our example, these are exactly the constraints we require; but compare this with a database that has persons, employees, and students. Both an employee and a student are special kinds of person, but we do not necessarily want either (a) or (b) to hold since a person could be neither a student nor an employee or could be both a student and an employee. In Pascal, we can relax constraint (a) by adding a third (empty) variant to the record type, but there is nothing we can do to relax (b). In Codasyl we can use a set that contains two member classes to suggest discriminated unions as we did in 3-1; but we need to constrain each set occurrence to have at most one member, and this can only be done by making sure that each updating program respects this rule. There is no checking mechanism built in to Codasyl to enforce this constraint.

The other new data model is simply that of an extensionally defined *function*. The relationship of a member to an owner class in Codasyl is many-one, as is the relationship

of a non-key to a key field in a relation. This leads to the possibility of expressing the database as a collection of functions so that, for example, *Name* can be thought of as a function from *Part* to *String*. Among the advantages of the functional data model is that extensional (database) functions can be given similar semantics to other functions. This makes the functional data model a natural adjunct for *functional* programming languages, in which functions are values and can be manipulated by other (higher order) functions. Requiring a language to be functional is therefore a natural step towards type completeness.

4.1. Daplex and Adaplex

The functional data model was first exploited in two languages, Daplex [Shipman 81] and FQL [Buneman 82]. The second of these has a polymorphic type system related to that of ML, which we will discuss later. Daplex is interesting because it also captures the notion of inheritance. Originally intended as a sublanguage, there was no intention that Daplex should posess any degree of type completeness or computational power. It is interesting to note, however, that Task 1, the data description, can be written in Daplex to capture all the constraints suggested in figure 2-6, Task 2 and Task 4 can be written as neatly as in any relational query language, and Task 3 can almost be written. Although Daplex contains a transitive closure operation, it is not quite powerful enough to include the arithmetic operations required here[17]

Rather than reviewing Daplex here (the reader is referred to [Kulkarni & Atkinson 85, Kulkarni & Atkinson 83] for experiments with the language), we shall look at a substantial endeavour to conflate Ada and Daplex in the language Adaplex [Computer Corporation of America 83]. Adaplex has maintained most of Daplex's data model in that it exploits the functional model and inheritance. Some of the computational structures of Daplex have been omitted since they can be performed in Ada. It should be emphasised that Adaplex is not an embedding of Daplex in Ada in the sense of embedded SQL. It is an extension of Ada to incorporate new data types and control structures corresponding to the functional model as formulated in Daplex, but consistent with the design and philosophy of Ada.

Although Daplex [Shipman 81] was a major influence in the design of Adaplex, the

[17] As in Aldat (section 3.7) there is a problem with limited forms of transitive closure. Task 3 was chosen because it is a natural task that shows up the problems of predefined transitive closure operations.

```
DECLARE Part () →» ENTITY
DECLARE Name(Part) → STRING
DECLARE Pno(Part) → INTEGER -- No facility for domains - like Pnum
DECLARE Supplier() →» ENTITY
...
DECLARE BasePart() →» Part
DECLARE UnitPrice(BasePart) → INTEGER
DECLARE Mass(BasePart) → INTEGER
DECLARE Sources(BasePart) →» Supplier
DECLARE CompositePart() →» Part
DECLARE AssemblyCost(CompositePart) → INTEGER
DECLARE MassIncrement(CompositePart) → INTEGER
DECLARE SubParts(CompositePart) →» Part
DECLARE Quantity(CompositePart, SubParts(CompositePart)) → INTEGER
DEFINE UsedIn(Part) →» INVERSE OF SubParts(CompositePart)
```

Figure 4-1: Task 1: A Daplex description of the parts data

notation for defining the database bears little resemblence to the Daplex notation for function declarations shown in figure 4-1, but the main changes are surface syntax to comply with Ada convention. For example, the double headed arrow of Daplex becomes **set of** in Adaplex. Figure 4-2 shows how the definition of the database (Task 1) is recorded in Adaplex. Unfortunately, unlike Daplex, Adaplex does not allow us to define a derived function, recording the fact that *WhereUsed(Part)* →> **INVERSE OF** *ComponentList(CompositePart)*.

The operation of extracting the expensive bought in parts is simply encoded in Adaplex as is shown in figure 4-3, obtaining *Name(p)*) automatically from *Part*, as the inheritance was defined by the **include** statement. Note that **By Descending** allows the simple specification of the order in which the selected entities are processed.

Coding Task 3 in Adaplex is more problematic. Figure 4-4 shows a program based on the algorithm illustrated first in Pascal (Figure 2-3). The recursive scan of the parts tree is much as it was in Pascal, and the management of getting function values specific to a specialisation of *Parts* is similar. The determination of which specialisation a part belongs to is different because the specialisation structure in Adaplex is more general, allowing overlap, and does not require the programmer to introduce the discriminator. One disadvantage of this is the loss of the possibility of encoding the selection using a case clause. The selection of a part given its name is much simpler in Adaplex than in Pascal. It would probably also be more efficient.

It is not certain whether a *Part* (Entity Instance) may be passed as a parameter as


```
database Parts is

  type Supplier;                                  --Needed for forward reference
  type CompositePart;

  type Pnum is 1..MaxInt;
  type Dollars is real;
  type Grams  is real;

  type Part is entity
    Pno          : Pnum;
    Name         : String(1..16);
    WhereUsed    : set of CompositePart;
  end entity;

  type BasePart is entity
    UnitPrice    : Dollars;
    Mass         : Grams;
    SuppliedBy   : set of Supplier;
  end entity;

  type CompositePart is entity
    AssemblyCost    : Dollars;
    MassIncrement   : Grams;
    ComponentList   : set of Requirement;
  end entity;

  include BasePart in Part;
  include CompositePart in Part;

  type Requirement is entity
    Uses         : Part;
    Quantity     : PosInt;
  end entity;
end Parts;
```

Figure 4-2: TASK 1: Adaplex data definition for parts

```
with Parts;
use Parts;
print_Expensive_Parts:
 atomic
   for each bPart in BasePart where UnitPrice( bPart ) > 100
         by descending UnitPrice( bPart ) loop
     PUT(Name( bPart ));
     PUT(UnitPrice( bPart ));
     PUT(Mass);
     NEW_LINE;
   end loop;
 end atomic
```

Figure 4-3: TASK 2: an Adaplex program to list expensive parts

shown in the above example, as Adaplex is quite restrictive about the places in which entity types may be declared and used.

For reasons given earlier, it is desirable to improve on the above algorithm by adding a data structure to hold a memo of subassembly properties already calculated. It is

```
--Using an algorithm similar to that in Pascal - figure 2-3

with Parts;
use Parts;
print_Cost_and_Mass:
 atomic
procedure costAndMass(in aPart:Part; out resultCost: Dollars;
                      out resultMass: Grams);
 declare
   subtotalCost: Dollars; subtotalMass: Grams;
 if aPart is in BasePart then
   resultCost := UnitPrice( aPart );
   resultMass := UnitMass( aPart )
 else
   resultCost := AssemblyCost( aPart );
   resultMass := MassIncrement( aPart );
   for each cmpnt in ComponentList( aPart ) loop
     costAndMass( Uses( cmpnt ), subtotalCost, subtotalMass );
     resultCost := resultCost + Quantity( cmpnt ) * subtotalCost;
     resultMass := resultMass + Quantity( cmpnt ) * subtotalMass;
   end loop;
 end if
end costAndMass;

declare itsCost: Dollars; itsMass: Grams;

costAndMass({aPart in Part where Name(aPart) = 'Mast '}, itsCost, itsMass);
PUT(itsCost);
PUT(itsMass); NEW_LINE;
end atomic
```

Figure 4-4: TASK 3: Calculating a part's cost and mass using Adaplex

unfortunate that this only appears to be possible if the data base designer has the forsight to include an appropriate entity type to hold these memos. Although such forsight is in general improbable, we assume it occured for our problem. The fragment of the parts database description then appears as in figure 4-5. The algorithm can now be modified to avoid reprocessing *CompositeParts* as shown in figure 4-6.

```
database Parts is

                        --type declarations as in fig 4-2

  type Memo is entity     --to store intermediate results
    ForPart               : Part;
    TotCost               : Dollars;
    TotMass               : Grams;
  end entity;
  unique Forpart within Memo;
end Parts;
```

Figure 4-5: TASK 1: Revised Adaplex data definition for parts

Note that the provision of sets in Adaplex makes it easy to introduce the appropriate data structure, and the assumed indexing mechanism makes it efficient. The extra effort required to construct this for Pascal, particularly if the number of parts allowed may be


```
--Using an improved algorithm similar to that in - figure 4-4
--Using a set to avoid retraversing common subassemblies

...                                              --Preamble as in fig 4-4
declare DonePart: set of Memo;

procedure costAndMass(in aPart:Part; out resultCost: Dollars;
                      out resultMass: Grams);
 declare subtotalCost: Dollars; subtotalMass: Grams; donePart: Memo;
 if aPart is in BasePart then
   ...                                           --As in fig 4-4
 else                                            --Not a base part - Composite
   donePart := { mem in DoneParts where ForPart(mem) = aPart };
   if count( donePart ) = 0 then                 --test if already memoised
     resultCost := AssemblyCost( aPart );        --Here for a new subassembly
     ...                                         --As in fig 4-4
     include(new Memo(ForPart => aPart, TotCost => resultCost,
                      TotMass => resultMass))
         into DoneParts;
   else
     resultCost := TotCost(donePart);            --here for a subassembly already
     resultMass := TotMass(donePart);            --traversed
   end if;                                       --End any Composite Part
 end if;
end costAndMass;

declare itsCost: Dollars; itsMass: Grams;
                                                 --set the memo empty
exclude doneParts from doneParts;

...                                              --Output as for fig 4-4
```

Figure 4-6: Revised Adaplex process to calculate cost and mass

very large[18] probably deters many programmers. Again there is some uncertainty as to whether a set may be declared at this point in the program, and also whether the coercion from a singleton set to the entity it holds works in this context. The use of an **exclude** statement to obtain an empty set is somewhat odd.

Note that the provision of persistence is associated with implicit entity classes, refered to as the extent of the entity class. In the examples, the use of an entity name after **in** (eg *Part*) is actually an abbreviation for the extent (eg *Part'extent*). These extents automatically persist in the module like objects introduced by the **database** construct. It is not clear whether entities, such as the instances of *Memo* in the revised version, will persist, but we hope they will not. Indeed that is why the set *DoneParts* was introduced rather than use *Memo'extent*. The introduction of the construct for transactions - introduced by **atomic** - is a major step in clarifying the nature of operations on persistent data.

[18] Expanding a hospital, aircraft or ship for example.

To do Task 4, record a new composite part, and how it is made, it turns out to be necessary to backtrack, and revise the definition of the parts database (Task1). This revision is shown in figure 4-7.

```
database Parts is
  ...          --As for declarations in fig 4-5
  type Ids is (NextPno, ... );
  type Id is entity
    IdName : Ids;
    value  : PosInt;
  end entity;
```

Figure 4-7: Adaplex declarations to keep track of part number allocation

Note, as in Pascal/R (see figure 3-14), the scalar value cannot be stored directly in the database; and the introduced packaging has the same overheads as before. Given this structure, Task 4 can be accomplished as shown in figure 4-8.

```
with Parts;    --according to revised def^n in fig 4-8
use Parts;

procedure makeSub(assembly:CompositePart; component:Part; q:Posint);
  include ( new Requirement (WhereUsed ⇒ assembly,
    Uses ⇒ component, Quantity ⇒ q) into ComponentList( assembly );
  end makeSub;

procedure Task4( ncp: CompositePart; newRequirements: set of Requirement);
  use the id in Id where IdName=NextPno;
  Pno( ncp ) := value;
  include (ncp) into CompositePart --automatically included in Part
  value := value + 1;

  for each r in newRequirements loop
    --handle exception for not found or ambiguous part
    makeSub( ncp, Uses( r ), Quantity( r ));
  end loop
end Task4;

declare newCompositePart: CompositePart; newComponentList: set of Requirement;

... --Code to conduct user dialogue and to create those values

task4:
  atomic
    Task4( newCompositePart, newComponentList );
  end atomic;
```

Figure 4-8: Task 4: recording the definition of a composite part

The use of set data structures to communicate the new values from outside the transaction, denoted by the atomic unit, means that the dialogue with the user can be prolonged, without the lock on the extents of *Parts* and *CompositeParts* etc. being of excessive duration. There is some doubt, however, as to whether entity typed variables, or set typed variables are permitted outside an atomic unit. The IS-A heirarchy

constraints are automatically met, as, when a new *CompositePart* is created with new, the corresponding *Part* entity is also created, and the program must provide values for its fields.

The exception mechanism of Ada can be neatly exploited in this sort of example. For example, the desetting mechanism, the, raises exceptions when its argument is not a singleton set.

It is slightly strange that the language does not permit multiple instances of databases of the same type, since corresponding ADA constructs can usually have multiple instances.

4.2. Taxis

Taxis[Mylopoulos *et al.* 80] represents the first attempt to exploit inheritance in a conceptual modelling language, but it is sufficiently close to a database programming language that we shall treat it as such. The basic notion in Taxis is a *class* which, roughly speaking, can be considered as an element of a type hierarchy. There is a most general class, *ANY_CLASS*, which has as subclasses: *VARIABLE_CLASS*, *AGREGATE_CLASS*, *FORMATTED_CLASS*, *FINITELY_DEFINED_CLASS*, *TEST_DEFINED_CLASS*, *EXCEPTION_CLASS*, *TRANSACTION_CLASS*. All database programming in Taxis is performed by defining classes. For example, what we have so far written as procedures is formed in Taxis by creating an appropriate instance of a *TRANSACTION_CLASS*, this being a predefined class whose intances may contain actions. A *VARIABLE_CLASS* is similar to a type with an associated extent, much like the entity types of Adaplex.

Classes are themselves objects in Taxis and instances of a *metaclass* called ANY. This is useful for attaching properties to classes themselves rather than to their instances. For example, to the class *PART* we might want to attach some statistics such as the number of objects in that class and the maximum number allowed. Figure 4-9 shows a meta-class definition for part with an asssociated transaction to count the number of parts in that class. Having done this, figure 4-10 shows the declarations for parts, base parts and composite parts.

A *VARIABLE_CLASS* is one that has a modifiable extent (i.e. insertions and deletions are allowed). In defining *BASE_PART* we are therefore creating a new *VARIABLE_CLASS*

```
VARIABLE_CLASS PART_CLASS with
    attribute_properties
        howMany : COUNT_OF;
        maxAllowed : INTEGER;
end

TRANSACTION_CLASS COUNT_OF returns INTEGER with
        parameters (c)
        locals
            c : PART_CLASS;
            ct : INTEGER;
        actions
            find : begin
                        ct <- 0;
                        for x in c do
                          ct <- ct + 1;
                   end
            give : return(ct)
end
```

Figure 4-9: A meta-class definition in TAXIS

```
PART_CLASS PART with
  keys
    partid: (Pno)
  characteristics
    Pno: {|1::1000|};
    Name: STRING;
  attribute_properties
    number_in_stock: INTEGER;
end

VARIABLE CLASS BASEPART isa PART with
  attribute_properties
    unit_price: DOLLARS;
    mass: GRAMS;
end

VARIABLE_CLASS COMPOSITE_PART isa PART with
  attribute_properties
    assembly_cost: DOLLARS;
    mass_increment: GRAMS;
end

VARIABLE_CLASS USE with
  characteristics
    where_used: COMPOSITE_PART;
    uses: PART;
  attribute_properties
    quantity: POSINT;
end
```

Figure 4-10: Data description in TAXIS

that inherits all the properties of *PART*. Within these declarations, a **key** specifies an attribute that must be unique within the class, and **characteristics** and **attribute-properties** specify respectively fixed and modifiable attributes. The class *USE* performs the usual many-many linking; here the separation of **characteristics** and

attribute_properties is somewhat arbitrary.

As it stands, this declaration ensures that every *PART* is either a *BASE_PART* or *COMPOSITE_PART* simply because the last two classes are the only subclasses of *PART* with modifiable extents. Therefore, the declaration ensures that the only way an object can become a *PART* is by being created as a *BASE_PART* or *COMPOSITE_PART*. The declaration does not, however ensure the mutual exclusion of these two classes. It is worth digressing here to note that it is quite possible that in a different situation, one might want to allow a part to be both base and composite, i.e. it is one that can both be bought or manufactured. Moreover, it might be desirable to attach further properties to such parts such as whether it is more desirable to manufacture them or buy them. Taxis allows us to create new classes that are subclasses of more than one existing class. This is known as *multiple inheritance*, which is not allowed in Simula or Smalltalk and would be difficult to model using the network or relational structures.

To ensure exclusion, assuming we want it, Taxis has an *EXCEPTION_CLASS*, which we could use to raise an exception whenever an attempt was made to create a *BASE_PART* that was already a *COMPOSITE_PART* and vice versa. Details of this are beyond the scope of this paper, but it should be emphasised that the entire structure of Taxis is built around classes and inheritance. Thus, just as a *VARIABLE_CLASS* can have more specialised subclasses, so can a *TRANSACTION_CLASS* and an *EXCEPTION_CLASS*.

Classes are objects in Taxis, therefore we should be able to create a metaclass *SUPPLIER_CLASS* and add, as an attribute of *BASE_PART*

```
attribute_properties
  ...
  suppliers: SUPPLIER_CLASS
```

which indicates that the *suppliers* of this part are given by the extent of a class which is in the meta-class *SUPPLIER_CLASS*. Unfortunately, there is a restriction that classes cannot be created dynamically which prevents us doing this and to describe the relationship between **suppliers** and *base_parts* we would have to resort to another "linking" class like *USE*. Contrast this with Adaplex, which provides **set of**.

Making classes objects of the same status as other values such as numbers etc. brings up the delicate the relationship between *data type* and *class*. It is clearly very convenient in database programming to have extents associated with certain types. In

contrast to Pascal/R, which maintains a strict separation between types and sets of values of that type, Adaplex and Taxis both have a construct that looks like a type declaration but has an associated extent. Moreover, it is these types with associated extents which are peristent. We do not advocate this singling out of a subset of types as privileged to have persistence, and believe that the lack of a mechanism for dynamically creating (possibly transient) subsets of those extents will lead to recurring problems. Galileo (see section 4.3) allows the type and extent to have different names, but maintains an association between the two. In Taxis, we should regard nearly everything as persistent since one should really think of a Taxis program as defining a permanent collection of classes; in fact it appears that there is very little non-persistent data in Taxis. Local variables of transactions are an example.

To show a simple transaction, figure 4-11 shows an implementation of TASK 2. I/O and the full range of iterative control structures are not defined in [Mylopoulos *et al.* 80], but we understand that they are taken from Pascal and our examples have been written accordingly. It is not clear how this transaction would, in practice, be invoked; presumably some model of user interaction within the framework of Taxis is required [Pilote 83].

```
TRANSACTION_CLASS EXPENSIVE_PARTS with
actions
  for a_part in BASEPART do
      if a_part.unit_price > 100 then
          writeln(a_part.name, a_part.unit_price, a_part.mass)
end
```

Figure 4-11: Task 2: An expensive parts transaction in TAXIS

A more comprehensive illustration of a transaction in Taxis is shown in figure 4-13. Note that a transaction has special attributes for parameters, locals, actions and results. Since we want this transaction to return pairs and, as in Pascal, there is no pairing operator, we have constructed a special class to describe records containing cost and mass fields. Notice that in this case we really only want this class to function as a type for a local variable. There is no need for the class *COST_AND_MASS_RECORD* to have an associated extent. The intention of this code should be reasonably obvious. Note that the transaction is recursively invoked by an expression of the form *x.cost_and_mass* where *cost_and_mass* is a name for the procedure taking one parameter. The result of this transaction is the value on this "attribute" of the part *x*.[19] We have not given a

[19] In the original TAXIS papers recursive transactions were not defined. We understand from Alex Borgida that the approach is now to treat only the outermost transaction as atomic, i.e. there are no partial commits.


```
AGGREGATE_CLASS COST_AND_MASS_RECORD with
  attributes
    cost: DOLLARS;
    mass: GRAMS;
end;

TRANSACTION_CLASS COST_AND_MASS(p:PART)
returns COST_AND_MASS_RECORD
  with
    locals
      a_part: PART;
      cmpnt: COMPONENT;
      total_cost: DOLLARS;
      total_mass: GRAMS;
      result_record: COST_AND_MASS_RECORD;
    action base_part_action on BASE_PART..cost_and_mass is
    begin
      total_cost → a_part.unit_price;
      total_mass → a_part.unit_mass;
      return(||cost: total_cost, mass: total_mass||)
    end;
    action composite_part_action on COMPOSITE_PART..cost_and_mass is
    begin
      total_cost → a_part.assembly_cost;
      total_mass → a_part.mass_increment;
      for cmpnt in USE do
        if cmpnt.whereused = a_part then
          begin
            total_cost → total_cost +
                cmpnt.uses.cost_and_mass.cost * cmpnt.quantity;
            total_mass → total_mass +
                cmpnt.uses.cost_and_mass.mass * cmpnt.quantity;
          end;
        return(||cost: total_cost, mass: total_mass||)
      end;
end
```

Figure 4-12: A recursive transaction in TAXIS

memoised version of this transaction, but see no particular difficulty in doing so. The appropriate "transient" data structure to hold partial calculations can be a class which is a local of some enclosing transaction. It is worth noting that it would be nice to have cost and mass as characteristics, rather than attributes of the *COST_AND_MASS_RECORD* class, but it didn't seem possible in a transaction to create an object with given attribute values rather than side-effecting the values of a local.

The update example shows a *TRANSACTION_CLASS* being used as a transaction. Suppose, for example, we tried to create a link to a sub-part whose name was not known in the data base. The attempt to **get_object** from *PARTS* would fail and abort the whole transaction including the generation of a new part number.

To summarise, Taxis has tried to exploit inheritance to describe all aspects of database programming. While we have some doubt as to whether this can be done without

```
TRANSACTION-CLASS ADD-PART  with
locals
  newName: STRING;
  newMI: GRAMS;
  newAC: DOLLARS;
  subPartName: STRING;
  subPartQuantity: INTEGER;
begin
...   //generate a new part number, NextPno.
            read in newName, newMI, newAC//
  insert_object aPart in COMPOSITE with
    name <- newName,
    assembly_cost <- newAC,
    mass_increment <- newMI,
    pno <- NextPno;
  repeat
    // read in subPartName and subPartQuantity //
    if q > 0 then
      begin
      get_object aPart from PARTS
        where aPart.name = subPartName
      insert_object in USE with
        uses <- aSubPart,
        whereused <- aPart,
        quantity <- subPartQuantity
  until subPartQuantity = 0
end
```

Figure 4-13: Task 4: An Update in TAXIS

making a large number of special classes with a consequent increase in complexity, we should also acknowledge that our tasks were not designed to illustrate all the uses of inheritance. The reader is referred to the source material for some more elegant demonstrations of Taxis [Mylopoulos *et al.* 80, Mylopoulos & Wong 80]. It should also be said that Taxis was originally intended as a language for *designing* database applications and therefore might be partially forgiven for certain deficiencies such as lack of i/o. If it is still intended for this purpose rather than for programming database applications, one wants to ask two related things of the language: first, how does one demonstrate that a given set of programs in some other language satisfies a Taxis design, and second, what guarantees does a Taxis design give on the correctness of an implementation.

A compiler for the full language is under development [Borgida 85a, O''Brien 83, Chung 84, Nixon 84] and a compiler from part of an earlier version of Taxis to Pascal/R was built at Toronto [Nixon 83]. Taxis is reported to be destined to be part of a major effort in programming environments, where *expert* tools will assist the programmer in developing an efficient information system in DBPL [Schmidt 85].

4.3. Galileo

Among the languages that have adopted a new type system, designed to make the meaning of the long term data more apparent, is Galileo [Albano, A. *et al.* 83, Albano *et al* 85a], which is a research language being developed at the University of Pisa. As such it is presented alongside Taxis and Adaplex. It has a type system partially derived from an early version of ML [Gordon *et al.* 79][20]. The derivation includes the addition of a class and subtype mechanism. These are used in the data description given in figure 4-14.

```
use PartsDB :=
 (type Unit ↔
  ( num drop nonfix *, nonfix mod, nonfix /, nonfix div
  with
    q:num * u:Unit := mkUnit(q * repUnit u )
 ) in
 type    Dollars ↔ Unit
 and     Grams ↔ Unit
 in
 rec     Parts                class
  Part ↔
    (    Name: string
         and Pno: num assert within (1, 100000)
         and WhereUsed := derived all c in CompositeParts with
                    this into Subpart of CmpntList of c
    )
  key (Pno)
 and     BaseParts partition of Parts with CompositeParts class
  BasePart ↔
    (    is Part
         and UnitPrice: Dollars
         and Mass: Grams
         SuppliedBy: var seq Supplier
    )
 and     CompositeParts partition of Parts with BaseParts class
  CompositePart ↔
    (    is Part
         and var AssemblyCost: Dollars
         and MassIncrement: Grams
         and CmpntList: var seq Use
    )
 and     Use ↔
    (    SubPart: Part
         and Quantity: num assert >= 0 and integral
    )
 and     Suppliers class
  Supplier ↔
    ( ... )
...
)
```

Figure 4-14: Task 1 coded in Galileo

[20] These examples are fairly free from type declarations as Galileo inherits the type inference mechanism of ML [Milner 78].

In that figure, the *PartsDB* is made to persist by being preceded by **use** which indicates that the bindings being defined are to be stored permanently. In this case the binding of *PartsDB* to a new environment, which is a set of (name, object:type) bindings. Only one environment exists at the outer level, which is extended implicitly by bindings preceded by **use**, but, Galileo provides environment manipulation facilities, such as: inner environments, overlapping environments, and restricted and derived environments (views?). Any environment may be established as the context for name interpretation for subsequent code, by using **enter**, as in figure 4-15.

In coding task 1, operators are removed from the built in type *num* to produce a new abstract data type *Unit* with dimensional properties. Two distinct types *Dollars* and *Grams* are then defined. These represent the properties of those units more closely than any of the types available in the preceding languages.

The name *Parts* identifies a *class*, which is a sequence of instances of the abstract type, separately named, in this case as **Part**. It is important to note that the designers have chosen to require separate names for the class, and the type of its members, in contrast to other languages. Members of a class can be explicitly created and deleted. Creation implies insertion into the class. It is not possible to create two classes over the same abstract data type, since reiteration of the data type name or definition, generates a distinguished abstract data type. Iteration over a class is shown in figure 4-15.

```
enter PartsDB;
All b in BaseParts with UnitPrice > 100
```

Figure 4-15: Task 2 in Galileo without projection

```
enter PartsDB;
for bPart in BaseParts with UnitPrice > 100 loop
  (print Name of bPart; print UnitPrice of bPart; print Mass of bPart )
```

Figure 4-16: Task 2 in Galileo projecting the result to required properties

BaseParts and *CompositeParts* are defined as mutually exclusive subclasses of *Parts* by the **partition of** construct. Galileo also has a **subset of** construct to describe overlapping subsets of a class and a **restriction of** construct to arrange automatic insertion into the subclass on the basis of a boolean expression over constant properties evaluated when the member is created. The types of a subclass must be specialisations of the types of a its superclass. They inherit properties - as shown by **is** *Part* - and insertion into a superclass is a consequence of insertion into a subclass, see figure 4-16.

The **and** construct arranges two declarations to be simultaneously effective, and consequently allows the additional fields to be introduced simultaneously with those inherited from the supertype after **is**, this is equivalent to *specialisation*. **var** indicates fields may be updated, otherwise fields retain their creation time value. Compare with *properties* and *attributes* in Taxis, or **c** as a prefix of a type in PS-algol (see section 6.1), to declare a "variable" that is updated *only once* as it is created. **assert** establishes a constraint, and **derived** arranges that a field is automatically computed when needed, rather than stored. It is possible that this language gives the most precise definition of the data, in terms of a well defined set of types, of all the languages we examined.

Figure 4-16 shows a solution for Task 2. Since the language is implemented to immediately execute statements unless they are within a function, the result would be returned immediately, as in a "traditional" query language. However, in this case, the language provides a continuum into arbitrarily complex programs.

```
enter PartsDB;
use rec type cAndM := (cost: Dollars and mass: Grams)
and costAndMass(aPart:Part):cAndM :=
 (
 if aPart alsoIn BaseParts then                    (here for a simple base part)
   newcAndM( UnitPrice of aPart, Mass of aPart )
 else
   (                                               (here for a composite part)
   var resultCost := AssemblyCost of aPart
   and var resultMass := MassIncrement of aPart
   in
    (
    for cmpnt in cmpnts of aPart loop
     (
     var tempcAndM := costAndMass( SubPart of cmpnt )
     resultCost := resultCost + Quantity of cmpnt * cost of tempcAndM
     resultMass := resultMass + Quantity of cmpnt * mass of tempcAndM
     )
    newcAndM( resultCost, resultMass )
    )
   )
 )
and findPart( name: string): Part :=
 get Part with Name of Part = name;

itscAndM := costAndMass( findPart( "Mast" ));
print cost of itscAndM; print mass of itscAndM
```

Figure 4-17: Task 3 in Galileo

Figure 4-17 shows the Galileo solution to Task 3. The language is recursive, and so the traversal of the tree is easily organised. A type **CAndM** is constructed to pass back the compound result. The procedure to find a part is trivial, as **get**, defined over classes, either returns a unique element, or raises an exception.

```
enter PartsDB;
                                        {a structure to hold the memo}
 memos class memo ↔
  ( thePart: Part and itsCost: Dollars and itsMass: Grams )
  key thePart

...                                     {cAndM and start of costAndMass definitions}
...                                     {test and handle base case}
 else                                   {composite case}
                                        {first test if memoised}
 previousResult := get memos with thePart=p iffails
  (                                     {here not in memo yet}
  ...                                   {calculate cost and mass as before}
  newmemo( aPart, resultCost, resultMass )
  )
 newcAndM( itsCost of previousResult, itsMass of previousResult )
...                                     {as before}
```

Figure 4-18: Memoisation of Task 3 in Galileo

The new fragment of code, to mememoise *costAndMass* is shown in figure 4-18. The **get** operation to select a *memo* from *memos* will **fail** if this part has not been previously evaluated. Then the alternative code after **iffails** catches the exception and organises the calculation. Thus provision of **get** suggests that Galileo has a general indexing structure, at least logically.

```
enter PartsDB;
... {code to set up new values by dialogue}
... {so that newName contains the name,
          newAC  the assembly cost
   etc}

Task4( nme: string, nAC: Dollars, nMI: Grams, nCL: seq Use )
 (
 nextPno := nextPno + 1                 {issue new part number}
 newCompositePart( nme, nextPno, nAC, nMI, nCL )
 )

                                        {now a transaction}
Task4(newName, newAC, newMI, newCL )
```

Figure 4-19: Task 4 in Galileo

The code for task 4 is sketched in figure 4-19. *nextPno* was in *PartsDB* and therefore persistent. *newCompositePart*[21] makes a *CopmositePart* and the corresponding *Part*, placing them in their subclass and class respectively, if they pass all the asserted constraints. The evaluation of expressions at the top level are transactions, so *Task4*(...) is a transaction. An exception is raised if it fails to commit, eg a constraint fails and is not handled. The *WhereUsed* information is derived and so the programmer does not have to

[21] A function that generates values of type *CompositePart*, which is created and given this name when *CompositePart* is declared as a type.

arrange its construction - a marked improvement over the other languages we have considered.

If the group at Pisa are sucsessful in implementing Galileo, it will certainly be an interesting persistent language. Recent work on the language and its implementation is described in [Albano *et al* 85b, Albano *et al* 85c].

5. Polymorphism and Database Programming

At first sight, languages like ML and Poly, which form the basis for this section, have little to do with database programming, since they support limited forms persistence, nor do they support a bulk data type (such as a relation) that is essential for database work. However, we believe that the idea of polymorphic type systems is sufficiently fundamental to the future development of database programming that examination of these languages is essential. It is worth noting that ML and Poly are interactive (in the sense that they have incremental compilers) statically typed programming languages - a property that they share with database query languages.

5.1. ML

Based on the typed lambda calculus, ML [Gordon *et al.* 79] has been developed to the point that it is now a practical programming language[22]. For our purposes, its most important property is its polymorphic type system that allows types to be freely paramaterised by other types. However, there are several other properties that are relevant to our discussion. In particular, ML is incrementally compiled. This means that developing an ML program consists mostly of coding and interactively compiling small programs, and from practical experience, most debugging takes place through interaction with the compiler (in particular the type-checker). For the most part, type declarations are not needed in ML as there is a type inferencing mechanism [Milner 79] that determines a type for each expression based on its environment.

ML also has an exception handling mechanism and a system of modules which is currently under development. The latter is likely to have very important consequences for database programming [Cardelli & MacQueen 85].

As an introduction to ML, figure 5-1 shows a function for concatenating two lists. Note the use of pattern matching (:: is infix *cons*) to bind parameters.

```
val rec append( nil, l) = l |
        append((x::r), l ) = x::append( r, l );
```

Figure 5-1: A simple ML function definition

In response to this input, the ML compiler will output a message to the effect that append is of type $(\alpha\ list)\times(\alpha\ list) \rightarrow \alpha\ list$, α being a type variable. Subsequent attempts

[22] It is undergoing rapid development, and there is an attempt to arrive at an agreed standard, so the reader is warned that versions exist with substantial differences from the version that was used to test these examples [Milner 83, Milner 84, MacQueen 85].

to evaluate the expressions *append*([1;2;3], 4) and *append*([1;2;3],["cat"]) would both fail *at compile time*, the first because 4 is not a list and the second because there is no type α for which [1;2;3] and ["cat"] are both of type *alpha list*. Given this degree of type checking, it is impossible -- apart from problems posed by lack of storage resources -- for *append* itself to fail at run-time.

To turn to our database tasks, we have noted that ML does not have data types appropriate to database work. Our first task is therefore to create some new types that might be generally useful. Figure 5-2 shows a set data type, where the (concrete) data type *Set* is used internally to implement the abstract data type *set*[23]. Note that the use of the higher-order function *reduce* has made many of the definitions quite compact.

```
abstype 'a set = Set of 'a list
with
 val emptyset = Set nil
 and rec choose(Set nil) = escape "choose" |
         choose(Set(x'::l)) = x'
 and insert(x, Set l) =
         let val rec ins nil = x::nil|
                     ins(x'::l) = if x=x' then x'::l else x'::ins l
         in Set(ins l) end
 and reduce f (Set nil) z = z |
       reduce f (Set x::l) z  = f( x, reduce f (Set l) z);
 val remove(x,s) =
    reduce (fun(x',s') . if x=x' then s' else insert(x',s')) s emptyset;
 val member(x,s) = reduce (fun(z,b). ((z = x) or b)) s false;
 val union(s1,s2) = reduce insert s1 s2;
 val difference(s1,s2) = reduce remove s1 s2;
 val intersect(s1,s2)=difference(s1,difference(s1,s2));
 val mapset f s = reduce (fun(x,s').insert(f x,s')) s emptyset;
 val filter p s =
    reduce (fun (x,s').if p x then insert(x,s') else s') s  emptyset;
 val foreach s f = reduce (fun(x,y).(f x;())) s ();
 val settolist( Set l) = l;
 val rec listtoset nil = emptyset|
       listtoset(x::l) = insert(x,listtoset l)
end;
```

Figure 5-2: A set abstract data type in ML[25]

Throughout the paper we have stressed that the kind of directed acyclic graph that we need to represent the parts explosion diagram is extremely common in database design, and we have complained about the difficulty of enforcing constraints that will, for example, ensure that it is acyclic. Figure 5-3 shows a somewhat laborious implementation of an abstract data type for a directed acyclic graph (*dag*). The assumption is made that

[23] It is conventional to use case in this way in ML.

[25] the character ' behaves as a letter in forming ML identifiers, thus *p*, *p'*, *p''*, *'p* etc are different unrelated identifiers. However, there is a convention of using identifiers beginning with *a* ' as type parameter names, and we adhere to that in these examples.

unique values are attached both to the edges and points (vertices) of the graph. The functions *pointset* and *edgeset* return the sets of values associated with the points and edges in the graph. These are of type *'pointType* and *edgeType* respectively. *downedges* and *upedges* both return sets of (edge-value, point-value) pairs found on the downward and upward edges attached to a point. *addpoint* ads a new point to the dag, and addedge adds a new edge, but fails if adding the edge would create a cycle.

```
abstype ('pointType, 'edgeType) dag = Dag of
  ('pointType × (('edgeType × 'pointType) list) × (('edgeType × 'pointType) list)) list
with val newdag = Dag nil;
val pointset(Dag l) = listtoset (map (fun(p,_,_)·p) l );
val edgeset(Dag l) =
    reduce union ss emptyset
      where val ss =
        listtoset (map (fun(_,r,_)·listtoset(map (fun(e,_)·e) r)) l)
      end;
val rec downedges(p, Dag nil) = escape "downedges"|
    downedges(p,Dag ((p',l,_)::ll)) = if p' = p then listtoset l
                                      else downedges(p, Dag ll);
val rec upedges(p, Dag nil) = escape "upedges"|
    upedges(p,Dag ((p',_,l)::ll)) = if p' = p then listtoset l
                                    else upedges(p, Dag ll);
val rec above(p1,p2,d) =
  if member(p1, pointset d) & member(p2, pointset d)
  then (p1 = p2) or
      reduce (fun((_,p),b) b or above(p,p2,d)) (downedges (p1,d)) false
  else escape "above";
val addpoint(p, Dag l) = if member(p, pointset(Dag l))
                         then escape "addpoint"
                         else Dag((p,nil,nil)::l);
val addedge(p1,e,p2, d) =
   if not(member(p1, pointset d) & member(p2, pointset d)) or
     member(e, edgeset d) then escape "addedge"
   else if above(p2,p1,d) then escape "addedge"
       else
        let val rec putin nil = nil|
                putin((p,l1,l2)::ll) =
            if p1 = p then (p,(e,p2)::l1,l2)::putin(ll)
            else if p2 = p then (p,l1,(e,p1)::l2)::putin(ll)
            else (p,l1,l2)::putin(ll);
          val Dag ll = d
          in Dag(putin(ll))
        end;
val show(Dag l) = l
end;
```

Figure 5-3: A directed acyclic graph type in ML

One can achieve something similar to a record type in ML with a further data type declaration such as that shown for *Part* in figure 5-4. Here we have used an ordinary (not abstract) type declaration since there is no reason for a program not to have access to the representation. The selectors are defined as functions on the type *Part* making this a rather clumsy version of the declarations we saw in Adaplex. Note that we have been able, through the use of reference types (ref) to obtain the distinction between constant

and updateable attributes also used in Taxis (see 4.2) and in PS-algol (see 6.1). Since the relationship between *Part* and *Use* records is to be completely expressed by our *dag* data type, the only point in creating a new *Use* data type is to preserve the mnemonic value of *Quantity*.

```
type Part = MkPart of
          (string ·
          (Base of int ref·int ref· supplier set ref |
           Composite of int ref · int ref)) ref
with val Name (MkPart(ref (theName,_,_))) = theName
  and UnitPrice(MkPart(ref(_,_,Base(ref thePrice,_,_)))) = thePrice
  and Mass(MkPart(ref(_,_,Base(_,ref theMass,_)))) = theMass
  and Supliers(MkPart(ref(_,_,Base(_,_,ref theSuppliers)))) = theSuppliers
  and AssemblyCost(MkPart(ref(_,_,Composite(ref theAC,_)))) = theAC
  and MassIncrement(MkPart(ref(_,_,Composite(_,ref theMI))))) = theMI
  and IsBase(MkPart(ref(_,_,Base _))) = true |
     IsBase(MkPart(ref(_,_,Composite _))) = false
end;

type Use = MkUse of int ref
with val Quantity(Mkuse (ref q)) = q
end;

type Supplier = ...

type DatabaseType = (MkDatabase of (((Part # Use) dag) # Supplier set)) ref
with val Diagram(ref (MkDatabase(theDAG,_))) = theDAG
  and Parts aDataBase = pointset (Diagram aDataBase)
  and Suppliers(ref MkDatabase(_,theSuppliers))) = theSuppliers
end;

val Database = MkDatabase(emptydag, emptyset);
```

Figure 5-4: The Database declaration (Task 1) in ML

Equality of reference types follows the same rules as, say, Pascal. Therefore we can insert two records in the database that contain the same attribute values (as in Adaplex, Galileo and PS-algol). However there is no associated extent with any type, and our final val declaration for the database is constructed in the same spirit as Pascal/R.

The code for the next two tasks in ML is extremely simple because we constructed the appropriate higher-level functions for sets. The code for Task 2 is sufficiently close to the simple query languages (further similarity could have been achieved by writing an infix *where* to replace *filter*) to justify our earlier claim that syntactic simplicity on simple queries should not exclude general computational power. ML comes close to an ideal; and designers of query languages should also consider the "Z-F" notation used in SASL [Turner, D.A. 81] and Miranda [Turner 85]. The query could have been even simpler if one had been able to extend the *print* function to work on sets and avoid the need for an explicit iteration. Perhaps the main clumsiness is the explicit reference to *database* that

is needed. One would like to open a syntactic scope, much as is done in Pascal/R.

```
foreach (filter IsBase (parts database))
        (fun p.if unitprice p > 100 then print(name p, unitprice p, mass p));
```

Figure 5-5: Printing expensive parts in ML

Memoising the code for Task 3 poses no particular problem in ML, but we should note that a generic "table" type similar to that of PS-Algol would be extremely useful (as it would be in Pascal) if one wants to write a momo function that is both general and efficient. One cannot implement this efficiently in ML as the implementation methods (hash coding or search trees) call for generic hashing functions or generic comparison functions. These are not available to the programmer, nor can they be implemented by the programmer without violating the type rules. Memoising also presents another interesting challenge to the designers of functional programming languages Ideally one would like to be able to define a higher order function that takes a function as argument and yeilds a more efficient version of the function as a result. It is easy enough to write down the rules for transforming the function, but to implement the transformation requires access to the structure of the function.

```
val rec costAndMass aPart =
 if IsBase aPart then (UnitPrice aPart, Mass aPart)
 else reduce (fun((aUse, thePart), (subTotalCost, subTotalMass))
     let val (resultCost, resultMass) = costAndMass thePart
     and q = Quantity(aUse) in
       resultCost*q + subTotalCost, resultMass*q + subTotalMass
     end)
     downedges( aPart, Diagram Database)
     (AssemblyCost aPart, MassIncrement aPart)
```

Figure 5-6: Function to compute cost and mass in ML

Updating the database is follows the original Pascal example. We assume that the update program is provided with a new (composite) part and a list of (*Use*, *Part*) pairs. It should be noted here that in this representation, the updating program should take care to ensure that the new part is composite since our type declarations do not prevent base parts at non-terminal points of the *Dag*. This program can be considered a transaction in the sense that the database is only upated at the end, and any failure (e.g. in adding to the *Dag*) would abort the function before this update took place. This is a somewhat facile notion of the implementation of a database transaction since we are calling for the database to be copied. Had we implemented our *Set* and *Dag* data types using side effects, ML would not be able to provide a mechanism for implementing this as a transaction.


```
val update(NewPart: Part, Sublist: ((Use×Part) list)) =
( let d = addpoint( NewPart, Diagram( Database ))
  in let val rec addlist( nil, d ) = d |
         addlist(( e, r ) :: l ) = addedge( NewPart, e, r, addlist( l, d ))
     in
       addlist( l, d )
     end
  end;
  Database:= ! MkDatabase( d, Suppliers( Database ))
)
```

Figure 5-7: Updating the database in ML

5.2. Poly

The language Poly [Matthews 85a], which is a derivative of Russel [Demers & Donahue 79], has been developed at Cambridge as a parametric polymorphic language. This form of polymorphism depends on the introduction of type parameters to procedures, which differs from that in ML (section 5.1). A discussion of the relationship between these forms of polymorphism is given in [Cardelli & Wegner 85]. A particular feature, allowing succinct programming, in Poly is the enclosure of type parameters to procedures in square brackets in the procedure declaration. This indicates that the actual parameter for this type may be omitted from the procedure application, as it may be inferred from some other actual parameter. The allowable actual values may be restricted, and the names of operations on the type may be statically introduced into the procedure's scope, by a signature, called a *specification* in Poly, introduced by `type`.

The language has four type constructors, `record`, `structure`, `union`, and `proc`. New abstract types may be declared using `type ... end`. The language includes exceptions. Iterators may be defined over new types, as in CLU [Liskov *et al* 81], from which Poly's notation for values associated with a type derives.

To show the power of this parametric polymorphism, we present fragments of a Poly program capable of doing all the tasks. However, it has been *internationalised*, that is, parameterised with the currency and mass units in use. Instances of the database type may then be used in different countries, but the type rules ensure that no computation accidentally mixes roubles and dollars etc. Figure 5-8 shows the majority of task 1, as the definition of a type for *Part*, parameterised over *Currency* and *Mass_Unit*. Figure 5-9 is then a procedure heading for a procedure, which, when applied to an actual currency, and an actual mass unit, will provide a *Part* type exclusive to those units.

Note that the [illegible] operators over a currency are defined precisely, and with

```
type (Part)                                  {Define the type of a part}
                                             {by its operations}
 make_part: proc(p_name: string)Part;
 convert_to_basic: proc( aPart: Part; cost: Currency; mass: Mass_Units;
                  supplier: Supplier_Type);
 convert_to_composite: proc( aPart: Part; cost_incr: Currency;
                  mass_incr: Mass_Units);
 add_part_to_composite: proc( cPart: Part; new_sub_part: Part;
                  quantity: integer);
 print: proc( Part );
 cost: proc( Part ) Currency;
 mass: proc( Part ) Mass_Units;
 form: proc( Part ) string;
                                             {introduce a pair type for cost and mass}
 cost_and_mass:
  type ( cost_and_mass )
   print: proc( cost_and_mass );
   cost: proc( cost_and_mass ) Currency;
   mass: proc( cost_and_mass ) Mass_Units;
  end;
 both: proc( Part ) cost_and_mass;
 over_all_parts:                             {define an iterator}
  proc()
   type ( Iter )                             {a standard type}
    init: proc Iter;                         {required standard values}
    continue: proc ( Iter ) boolean;
    value: proc ( Iter ) Part;
    next: proc ( Iter ) Iter
   end
end
```

Figure 5-8: A complete parameterised type for Part in Poly

```
let Supplier_Type == type ... end;

let p_Part ==
 proc( Currency, Mass_Units:                 {Two type parameters}
  type ( aUnit )                             {which must have defined}
                                             {at least the following}
   + : proc Infix ( aUnit; aUnit ) aUnit;
   * : proc Infix ( aUnit; integer ) aUnit;
   print: proc( aUnit )
  end
 ) ...                                       {the output type}
                                             {as in fig 5-8}
...                                          {the body of p_Part}
```

Figure 5-9: Poly procedure heading for the part type creator

reasonable legibility (compare with Galileo's the only other language in which this was feasible - section 4.3), but here, also with generality. Figures 5-10 and 5-11 show applications of this procedure to appropriately defined units.

Given these various type definitions, it is necessary to be explicit about the actual operator or procedure being applied, this is done by qualifying the name with the type name as in *type_name$value_name*. This can lead to a little verbosity, as can be seen


```
let Dollars ==
 type ( Dollars ) extends integer;
                                             {+ is inherited from integer}
  let * == proc( x: Dollars; y: integer ) Dollars
   begin
    x Dollars$* Dollars$up( y )
   end;
  let print == proc( x: Dollars )
   begin print( '$' ); print( x ) end
                                             {of defining Dollars}
 end

let Pounds ==
 type ( Pounds ) extends integer;
                                             {+ is inherited from integer}
  let * == proc( x: Pounds; y: integer ) Pounds
   begin
    x Pounds$* Pounds$up( y )
   end;
  let print == proc( x: Pounds )
   begin print( x ); print( 'lb' ) end
                                             {of defining Pounds}
 end

let American_Part_Type == p_Part( Dollars, Pounds )
```

Figure 5-10: Defining a part type for the American market

```
let Yen ==
 type ( Yen ) extends integer;
                                             {as for Dollars in fig 5-10}
  ...
   begin print( x ); print( 'yen' ) end
 end                                         {of defining Yen}

let Grams ==
 type ( Grams ) extends integer;
                                             {as for pounds in fig 5-10}
  ...
   begin print( x ); print( 'Grams' ) end
 end                                         {of defining Grams}

let Japanese_Part_Type == p_Part( Yen, Grams )
```

Figure 5-11: Defining a part type for the Japanese market

```
let Task2 == proc
 begin
  for ( American_Part_Type$over_all_parts(),
      proc(a_Part: American_Part_Type)
       begin
        if a_Part.form = "Basic part"
        then if a_Part.cost > dollars$100
        then print(a_Part)
       end);
 end;
```

Figure 5-12: Poly prints expensive parts

in the code for task 2, figure 5-12[26].

[26] Note that most of the time the compiler is able to disambiguate the overloading itself.

Note the use of the iterator. Of course the simplicity of this code depends on suitable definitions being made in the body of the procedure *p_Part*, as it did in ML and Galileo (sections 4.3 & 5.1). Later, PS-algol takes a similar approach (section 6.1). However, it is not simply a case of moving the work from one place to another. Generally, associating the code with the database or the type definition should allow it to be reused readily from many programs using the database or type. This has two advantages. The effort of writing the code is amortised over many applications, and, probably more importantly, the detailed operations on the data are localised to a known collection of code. This obviously has advantages for maintenance and integrity.

```
let Task3 == proc(a_Part: American_Part_Type)
 begin
  print( a_Part );
  print( a_Part.both );
  print( '\n' )
 end;
```

Figure 5-13: Poly code for Task 3 on an American part

Again the code, shown in figure 5-13, for Task 3 rests heavily on the code within *p_Part*'s body. Figures 5-14 and 5-15 show a fragment of this code. It uses essentially the same approach as ML (figures 5-3 & 5-6). We show only the memoised version, which was fairly easily constructed, as the implementation had a "generic"[27] hashing function *rehash*.

The update operation is straightforwardly coded, and is not shown, as again it is predominantly code written in the body of *p_part*. As described in more detail in the context of PS-algol, this gives protection. In both these languages, local variables in the type constructing body hold such things as the part number and the universe of parts. Each application of *p_part* creates a new environment for the body with a separate and totally independent set of these values.

Persistence is achieved in Poly [Matthews 85b] using the same loading algorithms as in PS-algol. The naming is however more convenient, as the code is statically bound to preserved, and distinguished outer environments (a similar treatment is planned in Galileo - section 4.3). This static binding means that a procedure needs to be compiled in each environment in which it is to be used. Avoiding this, yet keeping the simple naming and type equivalence, is an unresolved challenge. A solution has to be found

[27] provided the key was integer!


```
...                                                    {in the context of p_part's body}
                                                       {see fig 5-9}
let tran_close ==
 proc[r_type:                                          {an inferred type parameter}
  type (r_type)
   + : proc infix ( r_type; r_type ) r_type;
   * : proc infix ( r_type; integer ) r_type;
  end] (                                               {now required parameters}
  b_fun: proc( basic_part ) r_type;
  c_fun: proc( composite_part ) r_type ) proc( part ) r_type
 begin                                                 {tran_close body}
  let holder == rehash( 10, r_type )                   {start hash table empty}
  letrec t_c == proc( a_part: part ) r_type
   begin                                               {body of t_c}
    let source == a_part.source;
    if source.is_unset then raise cost_not_set
    else if source.is_basic then b_fun( source.proj_basic )
    else
     begin                                             {Composite part case}
      {look up part number in table, raise exception
       if not yet calculated}
      holder( repr( a_part.p_no ))
      catch proc( exception_name: string )r_type {catch not calculated yet}
       begin                                           {exception catcher}
        let composite == source.proj_composite;
        let value == fold(
         proc( a_rec: composite_part$made_rec; initial: r_type ) r_type
          begin {the proc to be applied by fold}
           ( t_c( a_rec.parts ) r_type$* a_rec.quantity ) + initial
          end, {of proc applied by fold}
         c_fun( composite ) ) ( composite.made_from );
        holder( repr( a_part.p_no )) := value;         {Cache it}
        value                                          {and return it}
       end                                             {of exception catcher}
     end                                               {of Composite part case}
   end;                                                {of t_c}
 t_c
end                                                    {of tran_close}
```

Figure 5-14: Generic transitive closure code in Poly

before the technique used by Poly would work for large and commercial systems, as has been argued elsewhere [Atkinson & Morrison 85]. The Poly system of persistence at present is essentially based on a workspace approach (see section 3.6), but with incremental loading. The incremental type-checking necessary in PS-algol, is of course avoided with this static binding approach, simplifying system implementation considerably.

```
...                                    {some time after decl[8] in fig 5-14}
                                       {still in p_part body}
                                       {first define a type to hold both
                                       cost and mass, and to have necessary
                                       operators}
let cost_and_mass ==
 type ( cost_and_mass ) extends record( cost: Currency; mass: Mass_Units );
 let + == proc( z, y: cost_and_mass ) cost_and_mass
  begin cost_and_mass$constr( z.cost + y.cost, z.mass + y.mass ) end;
 let * == proc( z: cost_and_mass; q: integer ) cost_and_mass
  begin cost_and_mass$constr( z.cost Currency$* q, z.mass Mass_Units$* q ) end;
 let print == proc( z: cost_and_mass )
  begin print( "cost = " ); print( z.cost );
   print( "mass = " ); print( z.mass ) end
 end;                                  {of defining cost_and_mass type}

                                       {now apply tran_close to get
                                       parallel computation of cost and mass}
let both == tran_close(
 proc(z: basic_part)cost_and_mass
  begin cost_and_mass$constr( z.cost, z.mass ) end,
 proc(z: composite_part)cost_and_mass
  begin cost_and_mass$constr(z.cost_incr, z.mass_incr) end );
```

Figure 5-15: Using the transitive closure code to calculate both cost and mass in one traversal

6. Persistent languages

The last two languages we shall review are PS-algol and Amber. These are distinguished from other languages we have seen in that they adopt a uniform approach to persistence. In both these languages, any value may persist; thus persistence is not determined by type. The only other language we have seen with this uniform approach is APL 3.6, but because of its scoping rules (among other things), APL does not maintain an entirely uniform approach to values.

6.1. PS-algol

The language PS-algol [PPRG 85] is a derivative of S-algol [Cole & Morrison 82, Morrison 79]. It is an experimental language designed to show that persistence can be provided orthogonally to type [Atkinson *et al.* 81, Atkinson *et al.* 83b, Atkinson *et al.* 83a, Atkinson and Morrison 84] and that graphically based human computer interaction can be supported by language features [Morrison *et al* 86, Morrison *et al* 85a, Morrison *et al* 85b, Atkinson & Morrison 86]. This latter property is important in the typical application programs that involve persistence, but it is not discussed further in this paper.

```
structure Part (                       !define a class of objects
  cpntr Extra;                         !To Composite or Basic Part
  pntr WhereUsed;                      !to list of uses
  cstring Name;                        !constant name
  cint Pno )                           !not logically necessary

structure BasePart(                    !refered to via Part(Extra)
  int UnitPrice;                       !in Dollars
  int Mass;                            !in Grams
  pntr Suppliers )                     !to list of suppliers

structure CompositePart (
  int AssemblyCost;                    !in Dollars
  int MassIncrement;                   !in Grams
  pntr CmpntList)                      !to list of Uses

structure Use (
  pntr theSuperPart;                   !to a Part
  pntr theSubPart;                     !to a Part
  int Quantity)                        !how many used

structure Supplier ( ... )
  etc.

structure List(                        !standard list cell
  pntr Hd, Next )
```

Figure 6-1: Task 1 in PS-algol

Figure 6-1 shows the type definitions in PS-algol. The structures used are similar to

those in the first Pascal example, figure 2-1, but the data is not as precisely described as the types of referends of pointers are not specified, and integer values cannot be constrained to a subrange. But it does discriminate between updateable and constant fields (prefixing the type with c for assign once fields, and using = rather than := for initialisation of assign once "variables"), a consistent treatment of the concept, already seen in Taxis (section 4.2) and Galileo (section 4.3). The absence of precision about referends can be exploited to write (unparameterised) polymorphic functions, as shown for list manipulating routines in figure 6-2. Compare these with the similar ML functions for manipulating sets (figure 5-2).[28]

The language is strongly typed, as the access to a field of a structure, with the construct *pointer-expression (field-name)*, is checked for type consistency. This can be achieved because *field-names* must be unique within the current scope (they are introduced by the structure definition) and hence they identify the expected type of pointer object (ie a member of that structure class). This is the only dynamic type checking used in PS-algol, all other type checks are performed statically. This mixture has proved invaluable in providing a persistence mechanism supporting system evolution, and in providing associative structures (Tables etc) defined and implemented in the language.

The list functions are saved in a database called *lib* by arranging that they are accessible from the root of that database, just like any other data. In figure 6-2 that accessibility is via a *table* (a polymorphic data structure implemented in PS-algol). The value of the root of each database (returned by *open.database*[29]) is a table; entries in that table are any structures the programmer chooses to put there. The first *open.database* also starts a transaction. On *commit* all the data reachable from a database's root is transfered to that database, the transaction is terminated, and a new one begins.

[28]The distinction between the various forms of polymorphism is important. In the *parameterised* polymorphism of ML it is impossible (as in Pascal) to confuse reference types. In PS-algol there is only one pntr type, so confusion of the referend structure types is possible but will generate a run-time error as soon as any access to the structure is attempted. This in turn should be compared with other languages and data base systems where no checking (static or dynamic) is done on pointer types, and where confusion of these types can lead to a chain of unintended actions leaving one at the mercy of the operating system to catch some low-level and apparently unrelated violation of run-time constraints. Parameterised polymorphism is clearly desirable but, as we shall argue in the concluding section, techniques for combining it with database programming are not yet understood.

[29]PS-algol allows dots within identifiers - dot is NOT an operator in PS-algol.


```
structure List(                                  !define a class of List objects
  pntr Hd, Next )

structure ListPack (                             !to hold list functions
                                                 !fields for hd,tl etc
  ...
  cproc(pntr, proc(pntr → bool) → pntr ) filtr;
  cproc(pntr, proc(pntr)) app;
  cproc(pntr, proc(pntr → pntr) → pntr) map )

                                                 !decls for hd, tl etc
...
let filter = proc(pntr l, proc(pntr → bool) cond → pntr)
  if l = nil then nil
  else
    if cond( l(Hd)) then
      List( l(Hd), filter( l(Next), cond ))
    else
      filter( l(Next), cond )

let appToList = proc(pntr l, proc(pntr) f )
  while l ≠ nil do
    begin f( l(Hd) ); l := l(Next); end

let mapList = proc(pntr l, proc(pntr → pntr) f → pntr)
  if l = nil then nil
  else
    List( f( l(Hd) ), mapList( l(Next), f ))

let db = open.database( "lib", "pw", "write" )
if db is error record do begin write "'nSorry lib in use"; abort end
let lp = ListPack(
                                                 !defns of hd, tl etc
  ...
  filter, appToList, mapList)
s.enter( "ListPack", db, lp )
let done = commit()
if done isnt errorRecord do write "'n list pack in database"
```

Figure 6-2: PS-algol program to set up a library of list routines

The types of the list functions are specified in the **structure** class definitions, and are recorded with the data on the database, to reuse the data an equivalent class must be declared. Consequently the type can be checked on reuse, and there is sufficient information for all function applications based on the values extracted from the structure, to be statically typed.

As the procedures are first class, and all reachable data is preserved, environments referenced by the procedure are preserved. This can be used to initialise a database as shown in figures 6-3,6-4. The first figure stores in the *lib* database a function which when applied initialises a parts database and provides a set of functions over it. The second figure shows one such application, and the result being put in a new database. The data in the environment of *initialisePartsDB* is replicated each time that procedure is applied, and the closures of the procedures in the result have this environment bound

```
  ...                                   !declarations of figure 6-1
  ...                                   !declaration of ListPack figure 6-2
structure PartsDBPack(
  proc( string → pntr ) GetPart;
  proc( proc(pntr→bool);proc(pntr)) ForSomeParts;
  proc( string, int, int, pntr) NewCompositePart;
  ...
  )
                                        !obtain the latest list routines
let lib = open-database( "lib", "pw", "write" ) .
if lib is error-record do ...
let listPk = s-lookup( "ListPack", lib )
let filter = listPk(filtr); let appToList = listPk( app );
let mapList = listPk( map )

let InitialisePartsDB = proc(→ pntr)
  begin
  let nextPno := 0                      !running count of part numbers
  let Parts := nil                      !list of all parts
  let ByName = table()                  !empty table indexes parts by name
                                        !construct a PartsDBPack as a result
  PartsDBPack(
    proc(string name → pntr)            !GetPart
      s-lookup( name, ByName),
    proc(proc(pntr → bool cond; proc(pntr action )
      appToList( filter( Parts, cond ), action ),
    proc( string newName; int newAC, newMI; pntr newCL )
      ...                               !see figure 6-6
    ...
    )
  end                                   !of InitialisePartsDB

structure IPDBPack( proc( → pntr IPDB)
s-enter( "IPDPPack", lib, IPDBPack(InitialisePartsDB))

let done = commit()
...
```

Figure 6-3: PS-algol program to create and store a parts DB initialiser

```
structure IPDBPack( proc( → pntr IPDB)

let lib = open-database("lib", "pw", "read")
if lib is error-record do ...

let partsDBInitialiser = s-lookup( "IPDBPack", lib)(IPDP)
let newPartsDB = partsDBInitialiser()

let partsDB = open-database( "PartsDB", "Eigg", "write" )
if partsDB is error-record do
  begin write "\nCan"t create new DB", abort end
s-enter("PartsDB", partsDB, newPartsDB)
let done = commit()
if done isnt error-record do write "\nNew parts DB ok"
```

Figure 6-4: PS-algol program to set up a new parts database

to them. This also means the variables in the environment are now protected, and other programs can only use them via the procedures provided.


```
let partsDB = open-database( "PartsDB", "Eigg", "read" )
if partsDB is error-record do ...
                                        !declaration of PartsDBPack as figure 6-3
...
let forSomeParts = s-lookup("PartsDB", partsDB)(ForSomeParts)
                                        !declaration of Part & BasePart fig 6-1
...
let expensiveBasePart = proc(pntr aPart → bool)
  aPart(Extra) is BasePart and aPart(Extra)(UnitPrice) > 100
let printPart = proc(pntr aPart )
  write "\n", aPart(Name), " ", aPart(Pno)

forSomeParts( expensiveBasePart, printPart )

                                        !no changes implies no need to commit
```

Figure 6-5: Task 2 in PS-algol

The program to accomplish task 2, shown in figure 6-5 uses these procedures, not having direct access to the variables. The database lib has been used to allow separate compilation of relevant utilities, figures 6-3 & 6-4. The usual type-checking and incremental loading of PS-algol, arranges the correct composition of precompiled components when they are first used. The present version of PS-algol does not have immediate execution, consequently it is not possible to make this appear like a conventional query. The program shown is dominated by the text to define and explicitly extract utilities from the database. Only definitions that are going to be used are given, so they form a subset view of the data, making it clear which parts of the database definitions each program depends on.

The program in PS-algol to do task 3 is very similar to that given for Pascal, except for superficial syntax, so it is not shown here. The composite value *cAndM* could be returned from the procedure *costAndMass* as a structure instance of Galileo. The memoisation of this function can easily be achieved by constructing a *table* for the duration of the evaluation. The function *GetPart* shown in figure 6-3, would be used to find the part whose properties are to be computed.

Figure 6-6 shows the procedure, omitted from figure 6-3 that creates a new composite part. Note that it is actually declared in the context of *InitialisePartsDB* and so arranges consistent changes to the data in and refrenced via that environment. As no other routines exist which can update that data, we can be sure (if the routines in that environment, that perform update, are correct) it will remain consistent. This procedure would then be obtained from *PartsDB* and used within a transaction in the manner already illustrated.

```
                                    !a procedure expression to be compiled in the
                                    !context shown in figure 6-3

proc( string newName; int newAC, newMI; pntr newCL)
 begin
 nextPno := nextPno + 1             !issue part number
 let aCompositePart = CompositePart( newAC, newMI, newCL )
 let aPart = Part( aCompositePart, nil, newName, nextPno )
                                    !place on list of all parts
 Parts := List( aPart, Parts )
                                    !put in the index
 s.enter( newName, ByName, aPart )
                                    !arrange where used list
 let recordUse = proc( pntr u )
  begin
  let user = u(thePart)
  user(WhereUsed) := List( aPart, user(WhereUsed) )
  end
 appToList( newCL, recordUse )
 end                                !of expression for NewCompositePart
```

Figure 6-6: A procedure to perform creation of a composite part to perform Task 4 in PS-algol

Two features of PS-algol have been exploited heavily in these examples: the ease of putting *any* data structure in a database, even though they may contain inter-database references; and the use of procedures to encapsulate data. This consistent treatment of data and program allows systems to be built incrementally, and the use of data structures to be controlled. It leads to a quite different style of system structure and programming, which we have tried to illustrate here, and which is why the division of the examples over tasks is not homomorphic with the examples in other languages. Syntactic support for database access, tables and rebinding with items in the database is obviously desirable, but currently is not provided. Research into methods of meeting this deficiency is reported elsewhere [Atkinson & Morrison 85].

6.2. Amber

Amber is an experimental programming language for personal computers developed by Luca Cardelli [Cardelli 84a]. Although it is derived from ML and Galileo, its treatment of persistence is such that it is appropriate to group it with PS-Algol. Like PS-Algol, Amber is a statically scoped language that also has built-in support for graphics. It also has a uniform persistence mechanism based on the use of *dynamic* values. In addition, the type system exploits inheritance, introduced in Taxis and Galileo, and extends this to work on higher-order (function) types. Concurrency in Amber is achieved through the use of *channels*, which are also typed values and are used for transmitting values between processes.

The aspects of Amber that we shall examine here are persistence and multiple inheritance. A *dynamic* value in Amber is one which carries its type description with it; such a value has type **dynamic**. Dynamic values can be coerced to any given type by a coercion operator, and should the types not match, the coercion fails. Thus the form of type-checking achieved is somewhat similar to that obtained in PS-Algol through the use of **pntr** types. For example, in both languages one can construct lists of dynamic (or pointer) values and then write a polymorphic procedure to reverse such lists. Again, this form of polymorphism should not be confused with the *parametric polymorphism* of ML.

Any dynamic type in Amber can be made persistent. To see how this works, figure 6-7 shows the type declarations for our database schema. Since Amber has no bulk (set or relation) type, the type declarations have been based on the implementation used in the original Pascal example.

The **rec** ... **and** ... **and** ... construction is for creating a tuple of mutually dependent types or values. These are bound to names by **type** ... , ... , ... for types (**value** for values). The notation $\ll n_1: T_1, n_2: T_2, \ldots \gg$ declares a record type and $[n_1: T_1, n_2: T_2, \ldots]$ a variant type. The symbol **:>** (instead of **:**) indicates an updateable field. There are no explicit pointer types.

In designing the types for the Amber example, we have brought the variant up to the top level. Our reason for doing this is to illustrate the workings of multiple inheritance in Amber; apart from this and the different method of indicating modifiable values, the type declarations follow closely those of the original Pascal example (figure 2-1). The whole declaration is enclosed in a module which exports the type declarations. The reason for exporting the value DataBaseTypeVal is described below.

Figure 6-8 shows the structure of a typical program that is required to modify a database. We have shown that part of the program that reads in a value from a file, checks that it has the correct type, produces a (non-dynamic) value for that type, and, later, writes an updated version of that value out to a file. To elaborate, using the type declarations in the *PartsDataBaseType* module, the program first reads in a value from the file *PartsDataBaseFile*, which is a dynamic value (since dynamic values are the only values that can persist). The next statement checks its type. There is a built-in type *Type* that is used to describe the type of a dynamic object. Objects of type *Type* are not themselves types; they are just values whose structure reflects the type structure of the language.

```
module "PartsDataBase"
 export
  type CompositePart, BasePart, AnyPart, UseList, PartList,
       Supplist, DataBaseType
  value DataBaseTypeVal: Type

type (CompositePart, BasePart, AnyPart, UseList, PartList,
      Supplist, DataBaseType)=

rec CompositePart is ≪Name : String, UsedIn :> UseList,
                      AssemblyCost : Dollars,
                      MassIncrement : Grams,
                      MadeFrom :> UseList≫

and BasePart is ≪Name : String, UsedIn :> UseList,
                 UnitPrice : Dollars,
                 Mass : Grams,
                 Suppliers :> SuppList≫

and AnyPart is [Base: BasePart, Composite: CompositePart]

and UseList is [nil : unit,
                cell : ≪Quantity : Int,
                       Uses :> AnyPart,
                       UsedIn :> CompositePart,
                       NextUses :> UseList,
                       NextUsedIn :> UseList≫]

and PartList is [nil: unit, cell : ≪cont : AnyPart, next : PartList≫]

and SuppList is ....

and DataBaseType is ≪Parts :> Partlist, Supplier :> Supplist≫


value DataBaseTypeVal =
   typeOf(dynamic ≪Parts = nil, Suppliers =nil≫)
```

Figure 6-7: Task 1: describing the data in Amber[30]

The function *typeOf* takes a dynamic object and returns a value of type *Type*. Thus in checking that the dynamic value has the correct type, we compare the value produced by *typeOf(newValue)* with the *typeOf* an exemplar dynamic type in the module in figure 6-7. It should be emphasised that values of type *Type* are only used as a guide. The subsequent coercion will fail if the type of the dynamic value does not match the given type.

Figure 6-9 provides a simple example of inheritance in Amber. The code for *CostandMass* is not given here but would be similar to that used in our previous examples

[30] Amber uses the Roman face for type and value identifiers, and italic for labels. This is not merely convention, but is semantically significant. The assumption is made that the language will be used on machines with bit map graphics and a what-you-see-is-what-you-get editor.


```
import "PartsDataBaseType"
 type CompositePart, BasePart, AnyPart, UseList, PartList,
      Supplist, DataBaseType

 value DataBaseTypeVal : Type

let newValue = import("PartsDataBaseFile") -- newValue has type dynamic
if typeOf(newValue) = DataBaseTypeVal then
        let DataBase = coerce newValue to DataBaseType
        do ...
           ... -- Perform updates
           ...
        do export("PartsDataBaseFile", dynamic DataBase)
else printString("PartsDataBaseFile corrupted")
```

Figure 6-8: An outline of an update program in Amber

```
value CostAndMass =
 rec CostandMass : AnyPart → ≪cost : Dollars, mass : Grams≫ is ...


type Part = ≪Name : String, UsedIn :> UseList≫

value NameOf = fun p : Part → String is p.name

value MostExpensiveComponent = fun p : AnyPart → BasePart is ...
```

Figure 6-9: Illustrating the use of inheritance in Amber

such as in 2-1 that are based on a similar data structure. More interesting is the type declaration for *Part*. The fields of Part match (both in label and type) fields that are present in *BasePart* and *CompositePart*. From this it is inferred that both *BasePart* and *CompositePart* are *subtypes* of *Part*. This means, for example, that any value that is of type *BasePart* is also of type *Part*. Now consider the functions *NameOf* and *MostExpensiveComponent*. *NameOf* is of type *Part* → *String* and therefore can also take any object whose type is *Part* as argument. Thus since any object of type *BasePart* is also of type *Part*, *NameOf* is also of type *BasePart* → *String* and *CompositePart* → *String*. On the other hand, *MostExpensiveComponent* returns a *BasePart* and any value of this type is necessarily of type *Part*. Thus *MostExpensiveComponent* is also of type *AnyPart* → *Part*. As a result, the expression

NameOf(MostExpensiveComponent a)

where *a* is of type *AnyPart*, is well-typed.

In general, there is a partial ordering ($\sqsubseteq$) on record types and if a function f is of type $\sigma \to \tau$ then it is also of type $\sigma' \to \tau'$ whenever $\sigma \sqsubseteq \sigma'$ and $\tau' \sqsubseteq \tau$. From this ordering we can infer a partial ordering on function types:

$\sigma \to \tau \sqsubseteq \sigma' \to \tau'$ whenever $\sigma' \sqsubseteq \sigma$ and $\tau \sqsubseteq \tau'$

In other words this ordering on functions is *anti-monotonic* (or *contravariant*) on the

argument type. Cardelli [Cardelli 84b] provides a semantics for this ordering and shows how inheritance combines naturally with functional programming. It should be noted that a symmetrical ordering happens for variant types. More recently [Cardelli & Wegner 85] it has been suggested that type hierarchies of this kind can be combined with the parameterised-polymorphism in ML and that a type checking algorithm exists, but type inferencing is, in general, no longer possible.

This ordering may seem problematical at first and should be compared with the orderings on classes in Taxis. However, it has recently been shown [Buneman 85] that if this form inheritance is carried down to the level of values, a natural typing of relations, and other data structures such as the tables of PS-Algol, may be achieved.

7. Conclusions

The impetus to write this paper was the authors' conviction of the importance of programming language design. At the outset of this paper we gave three criteria according to which we expected to compare various designs for database programming languages: data type completeness, persistence, and computational power. An obvious way to summarise would therefore be to produce a consumer guide based on these criteria; but there would be little point to this since most of the languages we have surveyed are not yet on the market and in many cases the designs are not complete. When a sufficient number of these languages are in a state that they can be used in anger, we will be in a better position to make more accurate comparisons and perhaps to influence subsequnt designs. Moreover we will be better able to understand the implementation problems that are common to database programming.

The importance of a survey like this is not to produce detailed comparisons and analyses but to identify some common research themes, or principles, that we have perceived in most of the languages we have discussed and which are surely in need of further work. To recapitulate, they are

1. The need for a *uniform* language. There should be no major linguistic barriers to the development of programs that are computationally complex.
2. The provision of a mechanism to control persistence that is independent of type.
3. A built-in abstract data type, or family of types, to represent the regularity of large volumes of data.
4. A polymorphic indexing or retrieval mechanism for efficient implementation. It is possible that this could be combined with, or subsumed by, (3).
5. Programs, or procedures should be typed objects and uniformly treated as values that may persist.
6. The type system must represent some form of inheritance.
7. The types, or modules, must permit some form of incremental, dynamic, binding.
8. As much static type-checking should be performed as is consistent with (7).
9. A notation should be provided to signify "variables", that recieve a value computed at the time they are created, but which cannot be updated subsequently.

In addition, there are equally important issues that we have not discussed here. These

include locking and sharing mechanisms, privacy control, transactions and recovery These topics deserve a separate survey in the context of database programming and wi certainly have equally difficult research areas.

We will briefly elaborate on these points here, but we should stress that although thi paper is about design, implementation is equally important. It took some ten years fo relational technology to develop to the point that relational systems could compete i efficiency with existing database management systems; and we are only now in a positio to understand the advantages and drawbacks of relational database programmin languages and to use these as a basis for the discussions of the next wave o programming languages. If the ideas examined here are to be taken seriously, it woul be a great pity if we have to wait another ten years in order to understand what further research is needed.

7.1. Computational Power

Since we had assumed that all languages would provide adequate power, we expected that this issue would be quickly laid to rest. Recall that Task 2 was introduced (see 2) to test if a simple query could be expressed simply, while Task 3 was intended to discover if a languages computational power was at all limited. While it is true that all the languages beyond the relational query languages appear to provide the power of a Turing machine, several omissions were discovered that might have been pinpointed by putting a language design out to independent tests. Initially it was not clear, for example, whether recursive transactions were possible in Taxis (see 4.2). In Adaplex(see 4.1), the lack of temporary **entity** types made the coding of the memoised version of Task 3 particularly cumbersome[31]. In general, all the computations we demanded appeared possible, but sometimes so awkward as to be impracticable; and the awkwardness was invariably associated with lack of type completeness.

We claimed in the introduction that we saw no reason why a language should not be both powerful and simple enough for the uninitiated user, and we believe that languages such as SASL and Prolog have convincingly demonstrated this for general computations in languages with simple type systems. In Galileo (see 4.3), we saw that simple operations such as Task 2, could be written as straightforwardly and succinctly as a

[31] We understand that the designs of both these languages have recently been changed to correct these failings.

relational query language. ML (see 5.1) shared this brevity, but we were also able to express Task 3, a much more complicated compuation, equally succinctly after designing the appropriate abstract data type to support general computations of that class.

The only problem we encountered with languages (such as ML) whose syntax is based on the lambda calculus is that the terseness of this syntax often means that what are really syntax errors give rise to incorrect programs that are syntactically well formed but ill-typed. The resulting type error messages that result, say, from a misplaced parenthesis or from forgetting the precedence of operators can be extremely confusing even for an experienced user. One of us has been involved in experiments [Kaplan 83] with Turner's "Z-F" [Turner, D.A. 81] notation for database iterations. Not only is it extremely simple to understand, being syntactically quite close to the relational query languages, it is also convenient for several forms of high-level optimisation [Nikhil 84].

Languages such as Galileo, ML, and Miranda suggest that it is now possible to produce, in languages with rich type systems, the same sort of smooth "ramp" for the user to climb in developing progressively more complicated programs that is exhibited by SASL, Prolog, and other simple interactive languages. It is important to continue this principle, especially where interaction with complex data types, such as database schemas becomes important. A major problem with these languages is that, while the treatment of simple *correct* programs may be simple, interpreting the type or run-time errors that result from *incorrect* ones often requires expert understanding of the language.

To summarise, we believe that there is no reason to limit the power of a database programming language on the grounds of simplicity, nor have we seen that there are gains in overall efficiency that result from allowing only a limited subset of operators. On the contrary, the inefficiencies introduced by having to switch language, program around the deficiency, or move data between programs, when the database language is not powerful enough for a given task, are much more serious. We suspect the same is true for the design of database machines. They should implement certain operations very efficiently, but should also interpret a language that is powerful enough to express an arbitrary database program.

7.2. Data Type Completeness

There is no need to elaborate any further on the failure of data type completeness. The challenge to the designer of a new or newly extended programming language is to produce the appropriate set of base types and type constructors. The constructors are are, of course, closely related to the choice of data model, whereas base types may be related to application targets, and both might be expected to vary. Looking at the more recent languages (see 4, 5 & 6) we find that there is considerable convergence in the choice of constructors. In particular, all have an indexing mechanism, which in some is directly exposed as a type; all have some notion of a *class*; and have attempted to deal with *inheritance*.

7.2.1. Indexes and Bulk Structures

The usefulness of a generic index type in programming languages is self-evident, and was the basis of several early database management systems. Provision of such structures is met with enthusiastic use. In PS-algol (section 6.1) and Poly(Section 5.2) the structure was insufficiently generic because the key type was restricted. In Pascal/R (section 3.2) the relation could be treated as a sparse, multidemensional, array and subscripted on a key tuple. Adaplex and Galileo do not make the structure explicitly available, but follow relational query systems in using it as an optimisation technique.

The design of these indexing constructs is invariably closely linked with the bulk data constructors such as relations. But this is only one context in which they may be useful, and it remains an open question as to whether they should have a separate type or whether both structures, relations and indexes, can be subsumed in one generic type.

Iterators and some collection of bulk operators for these structures are clearly essential, but what form should a "place holder" or "cursor" have, and how should the order of iteration be specified? 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. In particular, our experience has shown that the majority of "bulk" structures are small so that many implementation strategies have excessive overheads.

7.2.2. Classes

In Task 1 we wished to model the parts used in manufacturing as a set of uniformly typed entities. The need for this kind of representation is ubiquitous in databases and any database language must provide a mechanism for doing this. In Pascal/R, one starts by constructing a record type and using that to construct a relation. There is thus a clear separation between the underlying type of the entity and the associated extent (in fact there may be several extents associated with a given type). The **entity types** of Adaplex and *VARIABLE CLASSES* of Taxis do not make this distinction. Gallileo requires a seperate name for the extent and the underlying entity type, but although it will allow classes over identical types, the type for each class is a new type. Effectively, in all three, one declaration produces both the type and the extent. The designer of a new language should ask what is to be gained by doing this. In Pascal/R, the fact that the constituent fields of records that form the basis of relations are limited to certain types should be regarded as a failure of type completeness resulting from an engineering compromise. It would in fact be useful, to have a **relation of** *integer*, that is to have a generic *set* type.

If one is to adopt a single declaration for a type and extent, a consequent problem is whether the extent should necessarily be persistent. One may also require a representation of a subset of the extent. We have seen that temporary relations can be most useful in certain kinds of computation and that having persistence as a default will necessitate a separate method of declaring temporary extents.

Note that in Pascal/R, persistence is a property of a **database**, not of a **relation**. The fact that databases can contain only relations should again be regarded as an engineering compromise, and it would be extremely useful if objects of other types could put in a Pascal/R data base. The **database** constructor of Pascal/R is a good method of establishing a type and naming convention for a database. However, multiple instances, and fields of any type are required to conform with type completeness. But to conform with the need to allow the normal progress of database evolution, the binding cannot be entirely static. Amber (section 6.2 addresses this with its type **dynamic**, PS-Algol uses the dynamic properties of pointers and tables, and Galileo and Poly exploit the manipulation of environments. All these mechanisms address somewhat different functions and further research on the appropriate mixture of static and dynamic binding is discussed elsewhere [Atkinson and M...i... 84].

7.2.3. Inheritance

All the advanced database programming languages express some form of inheritance within their type system. Taxis is interesting in its attempt to use inheritance as the basis for nearly all aspects of the language. Each language takes a somewhat different approach to expressing constraints on specializations of a type. In Pascal (section 2.1), for example, the variants of a record are disjoint. Thus a *person* record type may have variants that correspond to student and employee types, but to create a *person* record type that represents both a student and an employee (or neither) requires extra variants to be added. In Adaplex, disjoint unions are the default, but can be relaxed with an **overlap** statement. Moreover a type is the union of its subtypes, and to allow that a *person* may be neither an *employee* or a *student* requires the declaration of a further empty subtype to account for this. By contrast, in Taxis, the default is that a *person* may be *student*, *employee*, both, or neither. In fact some work is required to express an integrity constraint that will prevent the two subtypes overlapping. In Galileo, three options are provided for specifying how subtypes and supertypes relate to each other. In Amber, the subtype relationship is inferred from the intrinsic properties of the type.

The interaction between procedures that operate on the type hierarchy also needs to be considered. We defined specializations of a Taxis *TRANSACTION CLASS* (that operated on parts) to operate on base- and composite- parts. This apparently gives rise to a hierarchy of transaction classes that corresponds to the hierarchy of other classes, or types. As we have seen in section 6.2, in Amber this hierarchy of procedure types is *contravariant* [Cardelli 84b] with the hierarchy of argument types since any procedure that takes a *part* as an argument can necessarily take a *base-part* as argument, but not vice-versa; in other words there are more procedures that operate on *base-parts* than on *parts*. This leads to the initially rather surprising conclusion that we should think of the type of a procedure that operates on *parts* as a specialization of the type of a procedure that operates on *base-parts*.

7.3. Polymorphism

The examples given in ML and Poly show that parameterised polymorphism is an extremely powerful tool for constructing new data types. However, there are a number of tasks relevant to database programming that still defeat polymorphic languages.

One that we discussed is the construction of a generic index type. For this one needs a hash function (for hash tables) or a comparison function (for search trees), and while it is

always possible to construct such functions, the construction usually depends on some system dependent information about the structures involved. Therefore, a generic index type, given the state of these languages, must be pre-defined.

A second, and more fundamental, problem relevant to databases is the data type of a relation. Suppose one wishes to assign a data type to the user-defined function:

$$join3(x,y,z) = join(x, join(y,z))$$

where *join* denotes the natural join of two relations. How would one express the argument and result types of *join3* if it is to be a generic function, i.e. defined for *all* relations (x,y,z)? The interaction between relational data types and polymorphic programming has yet to be properly understood; in fact we believe that the data type of the *natural join* has yet to be discovered.

As another example, consider the problem of substituting for every occurance of one value (substructure) in an data structure of arbitrary type another value of a different (or even the same) type. We can do this if we know the complete type of the data structure in which the substitution is to be done, but we cannot produce generic code to do this for all types, even though at some level of database restructuring, this is a common operation. Similar problems arise if one wishes to define a universal printing function, a snapshotting function, a generic forms based data acquisition function, etc.

The requirement for "universal application" programs like the problems just mentioned [Owoso 84, Owoso 85] is so extensive in the database context that this area requires immediate attention. One strategy, illustrated by the Poly examples, is to have every object provide a sufficient set of base functions. But if that is pursued, it will certainly be necessary to have a mechanism for adding to the set of base functions already in the database. This appears to be an unsolved problem for typed languages and the lack of a solution is still an argument that is often ranged against strict (or any) typing.

There are a number of programming tasks that require some form of self-reference in the language. We have remarked that it is not possible to write one function that memoises another in languages such as ML, while this is possible in an untyped language such as Lisp. A similar problem is that, in an incremental programming environment one would liker to call upon the compiler as one calls upon other functions, but how is the type of the compiler to be expressed?

Finally, we should emphasise that strict type-checking is highly desirable for database programming. The open question is how much of it can be made static? Since the term "static" is relative, this is only a question that can be answered with a better understanding of persistence.

7.4. Persistence

We have advocated througout this paper that persistence and data type should be orthogonal properties of data. We have also seen that transience is equally important for certain structures. Of the languages we have reviewed, PS-Algol, Galileo, Poly and Amber provide a completely uniform approach to persistence.

Some interactive languages provide a form of checkpointing (section 3.6). The user may save an existing workspace and recall it later on. However this is not adequate for database work. There is no way a user can exploit modularity to save parts of his workspace. For example it is likely that the database itself should be persistent, but that experimental programs should be kept in a separate, disposable, workspace. More important, when sharing of databases is required, checkpointing is completely inadequate. It is interesting to note here that APL does provide decomposable workspaces. However this is relatively easy to implement since APL's workspaces are *flat*. There is no need to maintain references from one workspace to another. Unfortunately, the use of flat workspaces prohibits the use of references - essential for database work - and makes static scoping difficult to manage.

Transactions and concurrency are both intimately connected with the provision of persistence. Transactions were explicitly supported by Adaplex **atomic** and in PS-algol. Other languages, which are addressing the problem of concurrency and distrobution find transactions essential [Liskov, *et al.* 83]. Recently, there has been considerable debate as to whether it is appropriate to build in a particular model of transactions, or whether it is better for the language to provide more primitive constructs, out of which the programmer constructs appropriate transactional and concurrent behaviour for the various abstractions he provides [Krablin 85, Weihl 85a, Weihl 85b]. This is still an open research question. Another open question is the treatment of exceptions in a persistent environment. In recent work, Borgida [Borgida 85b, Borgida 85c] has argued that it is necessary to design database management systems whose integrity constraints may be violated, and has analysed the persistence of error states in a database.

None of the languages, with the possible exception of Adaplex, which inherits the properties of Ada, has adequate facilities for concurrency. The design and implementation of concurrency constructs appropriate for database programming languages, is clearly a prerequesite for their serious use.

7.5. Modularity: a construct for persistence and organisation

Traditional programming language research has distinguished between the language itself and the programming language *environment*. One of the achievements of database programming has been to integrate parts of the environment with the language itself. The traditional form of database programming is, as we have seen, to treat the database as part of the environment and to have separate programs for compiling and linking schemas etc. In contrast, database programming languages have made an initial contribution to what must be desirable for all languages: a proper solution to the integration of languages with their environments. For databases, this requires a proper model of persistence and of how, and when, to perform binding and type-checking.

In programming environments, it is necessary to manipulate many persistent objects, such as: source code, interface specifications and compiled code. Such objects may be organised as modules[32]. It is reasonable to expect that a proper integration of any language with its environment will lead to the same database programming language being used for these modules as is used for all other dat, simplifying the programming environment.

Further, the modules provide a context for retaining the procedures and other data values, and for retaining the bindings between types, values and names. Indeed one of their roles is to assist in managing the large numbers of names that appear in a substantial software system. They also form a natural unit for component replacement. Consequently they are important in organising the construction and maintenance of both the body of data and the suite of programs. Indeed we expect to see any distinction betwwen the treatment of program and of other data disappear.

As we have seen in Poly, Galileo and PS-algol, where the analogues of modules were procedure closures or environments, that the module can be used to enforce arbitrary protection and constraints. This is the subject of much current research, for example

[32] In consequence, we recognise that all operating system command languages are of necessity database programming languages.

[Atkinson & Morrison 85, Cardelli & MacQueen 85]. If database programming language research succedes, then programmers will no longer *conciously* use databases, and it is possible that the form longer term data will be presented in, will be modules that happen to retain their data longer than others.

8. Acknowledgements

Much of the material for this paper was assembled during a seminar course at the University of Pennsylvania in the spring of 1984. We very much appreciate the contributions made by the participants in that seminar, Mark Reinhold, Sharon Perl, Rishyur Nikhil and others. We particularly appreciated the assistance given by Dave Matthews, who actually programmed all the tasks before our very eyes, in Poly. Many useful discussions with Luca Cardelli helped in our understanding of Galileo and Amber. Alex Borgida gave us much help in understanding Taxis. Joachim Schmidt and Mattius Jarke explained Pascal/R and its descendants, Modula/R and DBPL to us. Steve Fox of Computer Corporation of America, helped us with the Adaplex examples. Anthonio Albano and Renzo Orsini have helped us with Galileo. Ron Morrison, Tony Davie, and the rest of our Persistent Programming Research Group, helped by being both a critical and a constructive sounding-board. David MacQueen, Peter Wegner, and Bob Harper, as well as all the others that were at the Appin workshop, not yet meantioned, contributed to our general understanding of types.

Our work together was supported by our two Universities of Glasgow and Pennsylvania. The University of Edinburgh helped with access to equipment, during one stage in the preparation of the paper. Our work was funded by the British Science and Engineering Research Council, who gave a fellowship to Peter Buneman to spend a year in Scotland (GRC 86280). Support was also provided by ONR contract 5-20680 and a National Science Foundation CER grant 5-22030. Other grants from SERC: GRA 86541, GRC 21077, GRC 21060 and GRC 00000 provided travel, electronic communication and equipment.

References

[Aho & Ullman 79]
Aho, A.V. and Ullman, J.D.
Universality of Data Retrieval Languages.
In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 110-120. ACM, 1979.

[Ait-Kaci & Nasr 85]
Ait-Kaci, H. and Nasr, R.
Login: A Logic Programming Language with Built in Inheritance.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors), *Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16*, pages 000-000. Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.
Also to appear in ACM Principles of Programming Languages 1986.

[Albano *et al* 85a]
Albano, A., Cardelli, L. and Orsini, R.
Galileo: A Strongly Typed Interactive Conceptual Language.
ACM transactions on Database Systems 10(2):000-000, March, 1985.

[Albano *et al* 85b]
Albano, A., Giannotti, F., Orsini, R. and Pedreschi, D.
The type system of Galileo.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors), *Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16*, pages 175-195. Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.

[Albano *et al* 85c]
Albano, A., Ghelli, G. and Orsini, R.
The Implementation of Galileo's persistent Values.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors), *Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16*, pages 197-208. Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.

[Albano, A. *et al.* 83]
Albano, A., Cardelli, L. and Orsini, R.
Galileo: A strongly typed, Interactive Conceptual Language.
Technical Report, Bell Laboratories, Bell Telephone Laboratories, Internal Technical document Services, Murray Hill 1B-509, NJ, USA, 1983.
Submitted to ACM transactions on Database Systems.

[Amble *et al.* 79] Amble, T., Bratbergsengen, K., and Risnes, O.
ASTRAL: A structured and unified approach to database design and manipulation.
In *Proceedings of the database architecture conference, Venice, Italy.*, June, 1979.

[Astrahan 76] Astrahan, M. M. *et al.*
System R: relational approach to database management.
ACM transactions on Database Systems 1:97-137, June, 1976.

[Atkinson 78] Atkinson, M. P.
Programming languages and databases.
In S. B. Yao (editor), *The fourth international conference on Very Large Data Bases, Berlin, West Germany*, pages 408-419. September, 1978.

[Atkinson & Morrison 84]
Atkinson, M.P. and Morrison, R.
First Class Persistent Procedures are enough.
In *Proceedings of the fourth Conference on the Foundations of Theoretical Computer Science and Software Technology, Bangalore, India*, pages 223-240. Springer-Verlag, Berlin, December, 1984.

[Atkinson & Morrison 85]
Atkinson, M.P. and Morrison, R.
Types, Bindings and Parameters in a Persistent Environment.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors), *Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16*, pages 1-24. Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.
A revised version may be prepared for a paper at IFIP86 in Dublin.

[Atkinson & Morrison 86]
Atkinson, M.P. and Morrison, R.
Towards an Integrated Persistent Graphical Programming Language.
In *Proceedings of the 18th Hawaii International Conf. on Systems Sciences*, pages 000-000. January, 1986.

[Atkinson *et al.* 81]
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.
Also available as Tecnical Report CSR-94-81, Computer Science Department, University of Edinburgh.

[Atkinson *et al.* 83a]
Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R.
An Approach to Persistent Programming.
Computer Journal 26(4), November, 1983.

[Atkinson *et al.* 83b]
Atkinson, M.P., Chisholm, K.J., Cockshott, W.P. and Marshall, R.M.
Algorithms for a Persistent Heap.
Software Practice and Experience 13(7), March, 1983.

[Atkinson *et al.* 84]
Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.
Progress with Persistent Programming.
Databases - role and structure.
Cambridge University Press, Cambridge, England, 1984.

[Atkinson and Morrison 84]
Atkinson, M.P. and Morrison, R.
First Class Persistent Procedures.
ACM transactions on Programming Languages and Systems 7(4):000-000, October, 1984.

(BCS 81) The British Computer Society Query Language Group: Editor Samet, P.A..
Query Languages: A Unified Approach, Heyden and Son Ltd., 1981.
Monographs in Informatics.

[Borgida 83] Borgida, A.
Features of Languages for Conceptual Information System Development
Technical Report, Department of Computer Science, Hill Center, Rutgers University, New Brunswick, New Jersey 08903, USA, 1983.

[Borgida 85a] Borgida, A.
Communications regarding status and interpretation of Taxis.
Personal communication.
October, 1985

[Borgida 85b] Borgida, A.
Flexible data exceptions.
In Bubenko (editor), *VLDB11*, pages 000-000. August, 1985.
Title,editor,pages,organisation needed.

[Borgida 85c] Borgida, A.
Accomodating exceptions to type.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors), *Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16*, pages 265-271. Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotlan l, August, 1985.
Revision expected November 1985.

[Brachman 78] Brachman, R.J.
A New Paradigm for Representing Knowledge.
Technical Report BBN Report 3605, Bolt Beranek and Newman, Cambridge, MA, USA, 1978.

[Brachman 83] Brachman, R.J.
What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks.
Computer 16(10):30-35, October, 1983.

[Bragger *et al.* 83] Braggger, R.P., Dudler, A., Rebsamen, J. and Zehnder, C.A.
Gambit: An interactive Database Design Tool for Data Structures, Integrity Constraints and Transactions.
Database Techniques for Professional Workstations [Zehnder 83].
Eidgenossische Technische Hochschule Zurich, Institute Fur Informatik, 1983, pages 65-95.

[Brodie *et al.* 83] Brodie, M., Mylopolous, J. and Schmidt, J.
On conceptual modelling: perspectives from artificial intelligence, databases, and programming languages.
Springer-Verlag, Berlin, 1983.

[Buneman 82] Buneman, P., Frankel, R.E. and Nikhil, R.
An Implementation Technique for Database Query Languages.
ACM Transactions on Database Management 7(2), June, 1982.

[Buneman 85] Buneman, O.P.
Data Types for Data Base Programming.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors), *Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16*, pages 285-208. Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.

[Buneman *et al.* 82a]
Buneman, P., Hirschberg, J. and Root, D.
A CODASYL interface for Pascal and Ada.
In *Proceedings of the second British National conference on Databases: Bristol, England.* July, 1982.

[Buneman *et al.* 82b]
Buneman, O.P., Hirschberg, J and Root, D.
Integrating CODASYL with high level programming languages.
In *Proc. the 2nd of British National Conference on Databases.* 1982.

[Cardelli 84a] Cardelli, L.
Amber.
Technical Report, AT&T Bell Labs, Murray Hill, NJ, USA, 1984.

[Cardelli 84b] Cardelli, L.
A semantics of Multiple Inheritance.
Semantics of Data Types: International Symposium, Sophia-Antipolis.
Springer-Verlag, Berlin, 1984, pages 51-67.

[Cardelli & MacQueen 85]
Cardelli, L. and MacQueen, D.M.
Persistence and Type Abstraction.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors), *Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16*, pages 221-230. Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.

[Cardelli & Wegner 85]
Cardelli, L. and Wegner, P.
On Understanding Types, Data Abstraction, and Polymorphism.
ACM Computing Surveys 0(00):00-00, Aug, 1985.
Personal communication, submitted to Computing Surveys.

[Chen 76] Chen,P.P.S.
The Entity-Relationship Model: Towards a Unified View of Data.
ACM Transactions on Database Systems 11(1), March, 1976.

[Chung 84] Chung, K.L.
Implementation of Taxis, process management and enforcement of semantic Integrity Constraints.
Master's thesis, University of Toronto, Department of Computing Science, 1984.

[Clocksin & Mellish 81]
Clocksin, W.F. and Mellish, C.S.
Programming in Prolog.
Springer-Verlag, Berlin, 1981.

[Codd 70] Codd, E.F.
A Relational Model for Large Shared Databanks.
Communications ACM 13(6):377-387, 1970.

[Codd 79] Codd, E.F.
Extending the Relational Model of Data to Capture More Meaning.
ACM Transactions on Database Systems 4(4), December, 1979.

[Cole & Morrison 82]
Cole, A.J. and Morrison, R.
An Introduction To Programming With S-algol.
Cambridge University Press, 1982.

[Computer Corporation of America 83]
Smith, J.M., Fox, S and Landers, T.
ADAPLEX: Rationale and Reference Manual
second edition, Computer Corporation of America, Four Cambridge Center, Cambridge, Massachusetts 02142, 1983.

[Dahl & Nygaard 66]
Dahl, O. and Nygaard, K.
Simula, an Algol-based simulation language.
CACM 9:671-678, 1966.

[Date 81a] Date, C. J.
An introduction to database systems, 3rd edition.
Addison-Wesley, 1981.

[Date 81b] Date, C.J.
Referential Integrity.
In *The seventh international conference on Very Large Data Bases, Cannes, France.* VLDB, 1981.

[Date 83a] Date, C.J.
An Introduction to Database Systems.
Addison-Wesley, 1983.

[Date 83b] Date, C.J.
Database: a primer.
Addison-Wesley, Reading, Mass., USA, 1983.

[Demers & Donahue 79] Demers, A. and Donahue, J.
Revised report on Russell.
Technical report TR79-389, Cornell University, 1979.

[Fairbairn 82] Fairbairn, J.
Ponder and its Type system.
Technical Report 31, University of Cambridge Computer Laboratory, Cambridge, England, 1982.

[Fairbairn 85] Fairbairn, J.
A New Type-checker for a Functional Language.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editor), *Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16*, pages 107-123.
Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.

[Gallaire & Minker 78] Gallaire, H. and Minker, J., Eds.
Logic and Databases.
Plenum, New York, USA, 1978.

[Gallaire *et al* 84] Gallaire, H., Minker, J. and Nicolas, J-M.
Logic and Databases: a Deductive Approach.
Computing Surveys 16(2):153-185, June, 1984.

[Goldstein 80] Goldstein, I. P. and Bobrow, D. G.
Extending object oriented programming in Smalltalk.
In *Proceedings of the 1980 Lisp Conference*, pages 75-81. August, 1980.

[Gordon *et al.* 79] Gordon, M.J., Milner, A.J.R.G., and Wadsworth, C.P.
Lecture Notes in Computer Science. Volume 78: *Edinburgh LCF.*
Springer-Verlag, 1979.

[Hall 83] Hall, P.A.V.
Adding Database Management to Ada.
ACM SIGPLAN notices 13(3):13-17, April, 1983.

[Hammer 75] Hammer,M.M. and McLeod,D.J.
Semantic Integrity in a Relational Data Base System.
In *1st International Conference on Very Large Data Bases.* September, 1975.

[Horowitz & Kemper 83] Horowitz, E. and Kemper, A.
AdaRel: A relational Extension of Ada.
Technical Report TR-83-218, University of Southern California, Department of Computing Science, Los Angeles, California, USA, November, 1983.

[Ichbiah *et al.* 79] Ichbiah *et al.*.
Rationale of the Design of the Programming Language Ada.
ACM Sigplan Notices 14(6), 1979.

[Iverson 79] Iverson, K.E.
Operators.
TOPLAS 1(2):161-176, October, 1979.

[Jarke & Koch 82] Jarke, M. and Koch, J.
A Survey of Query Optimization in Centralized Database Systems.
Technical Report, Center for Research on Information Systems, New York University, November, 1982.
CRIS 44, GBA 82-73 (CR).

[Jarke & Koch 83] Jarke, M. and Koch, J.
Range nesting: a fast method to evaluate quantified queries.
In *Proceedings of the ACM SIGMOD international conference on Management of Data, San Jose, California.* Association of Computing Machinery, May, 1983.
Also as a technical report from: the Center for Research on Information Systems, New York University CRIS 40, GBA 83-25 (CR).

[Jarke & Koch 84] Jarke, M. and Koch, J.
Query Optimisation in Databse Systems.
Computing Surveys 16(2):111-152, June, 1984.

[Kaplan 83] Kaplan, H.
High Level Interfaces for Databases.
Master's thesis, University of Pennsylvania, Department of Computing and Information Science, 1983.

[Kent 78] Kent, W.
Data and Reality.
North-Holland, 1978.

[Kent 79] Kent, W.
Limitations of Record-based Information Models.
ACM Transactions on Database Systems 4(1), 1979.

[Kersten 81] Kersten,M.L. and Wasserman,A.I.
The Architecture of the Plain Data Base Handler.
Software-Practice and Experience 11:175-186, 1981.

[Koch *et al.* 83] Koch, J., Mall, M., Putfarken, P.,-Reimer, M., Schmidt, J.W. and Zehnder, C.A.
Modula/R report. Lilith version.
Technical Report, Eidgenossische Technische Hochschule Zurich, Institute Fur Informatik, 1983.

[Krablin 85] Krablin, G.L.
Building Flexible Multilevel Transactions in a Distributed Persistent Environment.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors),
Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16, pages 83-105.
Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.

[Kulkarni & Atkinson 83]
Kulkarni, K.G. and Atkinson, M.P.
Use of PS-algol to experiment with data models.
Software Practice and Experience 15(0):000-000, xxx, 1983.

[Kulkarni & Atkinson 85]
Kulkarni, K.G. and Atkinson, M.P.
EFDM: Extended Functional Data Model.
The British Computer Journal 28(0):000-000, xxx, 1985.

[Lampson 77] Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.L.
Report on the programming language EUCLID.
SIGPLAN Notices 12(2), 1977.

[Liskov *et al* 81] Liskov, B. *et al.*
Lecture notes in Computer Science. Volume 114: *CLU reference manual.*
Springer-Verlag, Berlin, 1981.
Goos and Hartmanis, Eds.

[Liskov, *et al.* 83]
Liskov, B., Herlihy, M., Johnson, P., Leavens, G., Scheifler, R. and Weihl. W.
Preliminary ARGUS reference manual.
Technical Report Memo 30, Programming Methodology Group, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts 02130, USA, October, 1983.

[MacQueen 85] MacQueen, D.M.
Modules for Standard ML.
Polymorphism 2(2), to appear, 1985.

[Matthews 85a] Matthews, C.J.
Poly Manual.
Technical Report 63, University of Cambridge, Computer Laboratory, Feb, 1985.

[Matthews 85b] Matthews, D.C.J.
Overview of the Poly Programming Language.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors),
Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16, pages 255-263.
Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.

[McCarthy 62] McCarthy, J., et al.
LISP 1.5 Programmer"s Manual.
MIT Press, Cambridge, Massachusetts, 1962.

[Merrett 77] Merrett, T.H.
Relations as Programming Language Elements.
Information Processing Letters 6(1):29-33, February, 1977.

[Merrett 83] Merrett, T.H.
Extending the relational data model to capture less meaning.
ACM SIGMOD RECORD , 1983.

[Merrett 84] Merrett, T.H.
Relational Information Systems.
Reston Publishing Co, Prentice-Hall, Reston, Virginia, USA, 1984.

[Merrett 85a] Merrett, T.H.
First steps to algebraic processing of text.
New Applications of Data Bases.
Academic Press, 1985.

[Merrett 85b] Merrett, T.H.
Persistence and Aldat.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editor),
Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16, pages 35-48.
Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.

[Merrett & Dächting 84]
Merrett, T.H. & Dächting, B.
Relational storage and processing of two dimensional diagrams.
Computers and Graphics 9(3), 1984.

[Michie 68] Michie, D.
'Memo' functions and machine learning.
Nature (218):19-22, April, 1968.

[Milner 78] Milner, R.
A theory of type polymorphism in programming.
Journal of Computer and System Science 17:348-375, 1078.

[Milner 79] Milner, R.
Flowgraphs and flow algebras.
Journal of the ACM 26(4):794-818, October, 1979.

[Milner 83] Milner, R.
A proposal for standard ML.
Polymorphism 1(3), December, 1983.

[Milner 84] Milner, R.
A Proposal for Standard ML.
In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*. ACM, 1984.

[Morrison 79] Morrison, R.
S-algol Language Reference Manual.
Technical Report CS/79/1, University of St. Andrews, 1979.

[Morrison *et al* 85a]
Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.
A Persistent Graphics facility for the ICL PERQ.
SPE 15(0):000-000, xxx, 1985.
Also available as PPR-10-84, Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland.

[Morrison *et al* 85b]
Morrison, R., Dearle, A., Brown, A.L. and Atkinson, M.P.
The Persistent Store as an Enabling Technology for Integrated Support Environments.
In *Proceedings of the 8th International Conf. on Software Engineering, London, England*, pages 000-000. IEEE, August, 1985.
Also available as PPR-15-85, Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland.

[Morrison *et al* 86]
Morrison, R., Dearle, A., Bailey, P., Brown, A.L. and Atkinson, M.P.
An Integrated Graphics Programming System.
The British Computer Journal 00(0):000-000, xxx, 1986.
Also available as PPR-14-85, Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland.

[Mycroft 84] Mycroft, A.
An inferential type system for Prolog.
Technical Report, Computer Science Department, University of Edinburgh, University of Edinburgh, Edinburgh EH9 3HD, Scotland, xxx, 1984.

[Mylopoulos & Wong 80]
Mylopoulos, J. and Wong, H.K.T.
Some features of the Taxis data model.
In *The 6th international conference on Very Large Data Bases, Montreal, Canada*. November, 1980.

[Mylopoulos *et al.* 80]
Mylopoulos, J., Bernstein, P.A. and Wong, H.K.T.
A Language Facility for Designing Database Intensive Applications.
ACM Transactions on Database Systems 5(2), June, 1980.

[Nikhil 84] Nikhil, R.
An Incremental, Strongly Typed Applicative Programming System for Databases.
PhD thesis, University of Pennsylvania, Department of Computing and Information Science, 1984.

[Nixon 83] Nixon, B.
A Taxis compiler.
Master's thesis, University of Toronto, Department of Computing Science, April, 1983.

[Nixon 84] Nixon, B. (ed.).
Taxis "84: Selected Papers.
Technical Report TR CSRG-160, University of Toronto, Computer Science Research Group, Toronto, Canada, June, 1984.

[O'Brien 83] O'Brien, P.
An Integrated Interactive Design Environment for Taxis.
In *Poc. of 1983 Softfair conf.*. Softfair, 1983.

[Olle 78] Olle, T.W.
The CODASYL approach to Data Base Management.
Wiley Intersience, New York, 1978.

[Owoso 84] Owoso, G.O.
Data Description and Manipulation in Persistent Programming Languages.
PhD thesis, EUCS, August, 1984.

[Owoso 85] Owoso, G.O.
Flexible data handling in Programming Languages.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors), *Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16*, pages 000-000. Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.

[Pilote 83] Pilote, M.
A Programming Language Framework for Designing User Interfaces.
ACM SIGPLAN notices 18(6):118-136, June, 1983.

[Pirotte 80] Pirotte, A. and Lacroix, M.
User Interfaces for Database Application Programming.
In *Infotech State of the Art Conference on Database*. Infotech Limited, 1980.

[PPRG 85] Persistent Programming Research Group, University of Glasgow, Department of Computing Science.
The PS-algol reference manual - second edition.
Technical Report PPR-12-85, Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, 1985.

[Rowe 85] Rowe, L.
Windows on Relations.
In Bubenko, J. (editor), *Proceedings of the eleventh international conference on Very Large Data Bases, Singapore*, pages 000-000. August, 1985.

[Rowe & Shoens 79] Rowe, L. and Shoens, K.
Data Abstraction, Views and Updates in RIGEL.
In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 71-81. ACM-SIGMOD, 1979.

[Schmidt 77] Schmidt, J.W.
Some High Level Language Constructs for Data of Type Relation.
ACM Transactions on Database Systems 2(3):247-281, September, 1977.

[Schmidt 85] Schmidt, J.W.
Plans for an Esprit project.
Personal communication.
August, 1985

[Schmidt & Brodie 83] Schmidt, J.W. and Brodie, L.M. (editors).
Relational Database Systems.
Springer-Verlag, 1983.

[Schmidt & Mall 83] Schmidt, J.W. and Mall, M.
Abstraction mechanisms for database programming.
ACM SIGPLAN notices 18(6), June, 1983.

[Shipman 81] Shipman, D.W.
The Functional Data Model and the Data Language DAPLEX.
ACM Transactions on Database Systems 6(1):140-173, March, 1981.

[Shopiro 79] Shopiro, J.E.
THESEUS - A Programming Language for Relational Databases.
ACM Transactions on Database Systems 4(4), December, 1979.

[Smith 77] Smith, J.M. and Smith, D.C.P.
Database Abstractions - Aggregation and Generalisation.
ACM Transactions on Database Systems 2(2), June, 1977.

[Stonebraker *et al.* 76] Stonebraker, M., Wong, E., Kreps, P., and Held, G.
The Design and Implementation of INGRES.
ACM transactions on Database Systems 1(3):189-222, September, 1976.

[Teitelman 75] Teitelman, W.
INTERLISP Reference Manual.
Technical Report, Xerox Palo Alto Research Center, Palo Alto ,California, USA, 1975.

[Tsichritzis 77] Tsichritzis, D. C. and Lochovsky, F. H.
Data base management systems.
Academic Press, 1977.

[Turner 85] Turner, D.A.
Miranda: A Non-strict Functional Language with Polymorphic Types.
In Jouannaud, J-P. (editor), *Functional Programming Languages and Computer Architecture*, pages 1-16. Springer-Verlag, Berlin, September, 1985.
S-V lect. notes in CS no. 201.

[Turner, D.A. 81] Turner, D.A.
The Semantic Elegance of Applicative Languages.
In *Proceedings 1981 conference on Functional Programming Languages & Computer Architecture, Portsmith, New Hampshire.*, pages 18-22. October, 1981.

[Ullman 82] Ullman, J.D.
Principles of Database Systems.
Pitman, 1982.
Second Edition.

[Ullman 85] Ullman, J.D.
Implementation of Logical Query Languages for Database.
ACM transactions on Database Systems 10(3):289-321, September, 1985.

[van Wijngaarden *et al.* 69] van Wijngaarden, A. *et al.*.
Report on the Algorithmic Language Algol 68.
Numerische Mathematik 14:79-218, 1969.

[Vassiliou *et al* 83] Vassiliou, Y., Clifford, J. and Jarke, M.
How does an expert system get its data.
In Schkolnick, M. and Thanos, C. (editor), *Proceedings of the ninth international conference on Very Large Data Bases, Florence, Italy*, pages 70-72. VLDB, November, 1983.

[Wasserman 81] Wasserman, A.I. and Booster T.W.
String handling and pattern matching in PLAIN.
Technical Report 50, Laboratory of Medical Information Science, University of California, San Francisco, San Francisco, CA 94143, USA, February, 1981.
Submitted for publication.

[Wasserman *et al.* 81]
Wasserman, A.I., Shertz, D.D., Kersten, M.L., Reit, R.P., and van de Dippe, M.D.
Revised Report on the Programming Language PLAIN.
ACM SIGPLAN Notices , 1981.

[Weihl 85a] Weihl, W.E.
Linguistic Support for Atomic Data Types.
In Atkinson, M.P., Buneman, O.P. and Morrison, R. (editors), *Proceedings of the Appin workshop on Persistence and Data Types Persistent Programming Research Report 16*, pages 145-173.
Persistent Programming Research Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland, August, 1985.

[Weihl 85b] Weihl, W.E.
Implementation of Resilient, Atomic Data Types.
TOPLAS 7(2), April, 1985.

[Wirth 71] Wirth, N.
The Programming Language PASCAL.
ACTA Informatica 1, 1971.

[Wirth 83] Wirth, N.
Programming in Modula-2: Second Edition.
Springer-Verlag, Berlin, 1983.

[Zehnder 83] Zehnder, C.A.(Ed.).
Database Techniques for Professional Workstations.
Technical Report 55, Eidgenossische Technische Hochschule Zurich, Institute Fur Informatik, CH-8092 Zurich, Switzerland, September, 1983.

Table of Contents

List of Figures

Bibliography

Copies of documents in this list may be obtained by writing to The Secretary, Persistent Programming Research Group, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland.

Atkinson, M.P.
'A note on the application of differential files to computer aided design', ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
'Programming Languages and Databases', Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the department as CSR-26-78).

Morrison, R.
S-Algol language reference manual. University of St Andrews CS-79-1, 1979.

Atkinson, M.P.
'Progress in documentation: Database management systems in library automation and information retrieval', Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as departmental report CSR-43-79.

Atkinson, M.P.
'Data management for interactive graphics', Proceedings of the Infotech State of the Art Conference, October 1979. Available as departmental report CSR-51-80.

Atkinson, M.P. (ed.)
'Data design', Infotech State of the Art Report, Series 7, No.4, May 1980.

Bailey, P.J., Maritz, P. & Morrison, R.
The S-algol abstract machine. University of St Andrews CS-80-2, 1980.

Atkinson, M.P. (ed.)
'Databases', Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
'Nepal - the New Edinburgh Persistent Algorithmic Language', in Database, Pergammon Infotech State of the Art Report, Series 9, No.8 (January 1982) - also as Departmental Report CSR-90-81.

Morrison, R.
Low cost computer graphics for micro computers. Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
'EDQUSE reference manual', Department of Computer Science, University of Edinburgh, September 1981.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
'PS-algol: An Algol with a Persistent Heap', ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as Departmental Report CSR-94-81.

Cole, A.J. & Morrison, R.
An introduction to programming with S-algol. Cambridge University Press, 1982.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
'Algorithms for a Persistent Heap', Software Practice and Experience, Vol.13, No.3 (March 1983). Also as Departmental Report CSR-109-82.

Morrison, R.
The string as a simple data type. Sigplan Notices, Vol.17,3, 1982.

Atkinson, M.P.
'Data management', in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
'CMS - A chunk management system', Software Practice and Experience, Vol.13, No.3 (March 1983). Also as Departmental Report CSR-110-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in Databases - Role and Structure, see PPR-8-84.

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
Databases - Role and Structure, CUP 1984.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University

of Pennsylvania. October 1982. To be published (revised) in the Workshop proceedings 1983, see PPR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
'Current progress with persistent programming', presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
'An approach to persistent programming', in The Computer Journal, 1983, Vol.26, No.4 - see PPR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983 - see PPR-2-83.

Morrison, R., Weatherall, M., Podolski, Z. & Bailey, P.J.
High level language support for 3-dimension graphics, Eurographics Conference Zagreb, Sept. 1983.

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, CUP 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Hepp, P.E. & Atkinson, M.P.
Tools and components for rapid prototyping with persistent data, to be submitted.

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D., Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
The Proteus distributed database system, proceedings of the third British National Conference on Databases, (July 1984).

Kulkarni, K.G. & Atkinson, M.P.
EFDM : Extended Functional Data Model, to be published in The Computer Journal.

Kulkarni, K.G. & Atkinson, M.P.
EFDM : A DBMS based on the functional data model, to be submitted.

Atkinson, M.P. & Buneman, O.P.
Database programming languages design, submitted to ACM Computing Surveys - see PPR-17-85.

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", printed in ACM TOPLAS (Oct. 1985) - see PPR-9-84.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", to be published in Software Practice and Experience, November 1984.

Morrison, R., Dearle, A., Bailey, P., Brown, A. & Atkinson, M.P.
"An integrated graphics programming system", to be presented at EUROGRAPHICS UK, Glasgow University, March 1986.

Morrison, R., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", presented at 8th International Conference on SE Imperial, (August 1985).

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Buneman, P.
"Data types for data base programming", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Owoso, S.O.O.
"On the need for a Flexible Type System in Persistent Programming

Languages", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Hepp P.E. and Norrie, M.C.
"RAQUEL: User Manual" Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following Ph.D. theses have been produced by member of the group and are available from The Secretary, Persistent Programming Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland.

W.P. Cockshott
Orthogonal Persistent, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those produced and those planned plus their status at 15 November 1985.

Copies of documents in this list may be obtained by writing to The Secretary, The Persistent Programming Research Group, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ.

PPR-1-83	The Persistent Object Management System	[Printed]
PPR-2-83	PS-algol Papers: a collection of related papers on PS-algol	[Printed]
PPR-3-83	The PS-algol implementor's guide	[Withdrawn]
PPR-4-83	The PS-algol reference manual	[Printed]
PPR-5-83	Experimenting with the Functional Data Model	[Printed]
PPR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples	[Printed]
PPR-7-83	EFDM - User Manual	[Printed]
PPR-8-84	Progress with Persistent Programming	[Printed]
PPR-9-84	Procedures as Persistent Data Objects	[Printed]
PPR-10-84	A Persistent Graphics Facility for the ICL PERQ	[Printed]
PPR-11-85	PS-Algol Abstract Machine Manual	[Printed]
PPR-12-85	PS-Algol Reference Manual - second edition	[Printed]
PPR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C	[Printed]
PPR-14-85	An Integrated Graphics Programming Environment	[Printed]
PPR-15-85	The Persistent Store as an Enabling Technology for Integrated Project Support Environment	[Printed]
PPR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985	[Printed]
PPR-17-85	Database Programming Language Design	[Printed]
PPR-18-85	The Persistent Store Machine	[Printed]
PPR-19-85	Integrated Persistent Programming Systems	[Printed]
PPR-20-85	Building a Microcomputer with Associative Virtual Memory	[Printed]
PPR-21-85	A Persistent Information Space Architecture	[In Preparation]
PPR-22-85	Some Applications Programmed in a Persistent Language	[In Preparation]

PPR-23-85 PS/Algol Applications Programs [In Preparation]

PPR-24-85 A Compilation Technique for a Block Retention Language [In Preparation]

PPR-25-85 Thoughts on Concurrency [In Preparation]

PPR-26-85 An Exception Handling Model in a Persistent Programming Language [In Preparation]

PPR-27-85 Concurrency in Persistent Programming Languages [In Preparation]

PPR-28-85 A Type Theory for Database Programming Languages [In Preparation]