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

Data Types for Data Base Programming

Persistent Programming
Research Project 30

Alan Dearle.

PREFACE

This is a revised version of the paper presented at the Persistence and Data Types Workshop, Appin, August 1985 - see PPRR-16-85.

DATA TYPES FOR DATA BASE PROGRAMMING

Peter Buneman[1]
Department of Computer and Information Science
University of Pennsylvania

1. Introduction

There are three data types that find ubiquitous use in data base programming. They are

- record types,
- index or table types, and
- set or relation types.

It is suggested in this paper that the three types can be derived from one underlying construct, a *partial function*, or *map* as we shall call it that is well-behaved with respect to inheritance. Moreover the behaviour of these functions provides a link with the substantial research on datbase theory [Ullman 82] which, sadly, has yet to have any great influence on the types of database programming languages although it is generally regarded as extremely important in data base design. Another reason for wanting to find a unifying framework for these types is to form a better understanding of how the *functional* data model [Shipman 81, Buneman 82] relates to the *relational* data model [Codd 70] and how these in turn fit with inheritance.

To take an example of how these various types are used in data base programming languages, one of the earliest such languages is Pascal-R [Schmidt 77], which extends Pascal with a *relation* data type so that one can declare for example,

```
type Employee =  record
                   EmpNo : Integer;
                   EmpName : array[1..30] of char;
                   ...
                 end;
var EmpRel : relation <EmpNo> of Employee;
...
```

There are various methods in the language for performing the usual operations of the relational algebra; in addition, it is possible to use a relation such as *EmpRel* as an index

[1] This research was carried out on a British Science and Engineering Council Fellowship at the University of Glasgow

that, given a key of type integer, returns a record of type *Employee*. Thus $EmpRel[1234]$ returns that record - if it exists - whose $EmpNo$ is 1234. In some sense the relation $EmpRel$ is doing double duty both as a set and as an index. More recent database programming languages such as Taxis [Mylopoulos *et al.* 80], Adaplex [Computer Corporation of America 83] and Galileo [Albano *et al.* 83] provide similar constructs although in some cases the index type is not directly available to the programmer but is nevertheless implemented as a method for optimisng the manipulation of relations or similar types.

Although these types are all implemented, they are not necessarily treated unifomly within the language. One reason is undoubtedly that two of these types, index and relation, may be single out as *persistent* types and the implementation of persistent types may restrict what can, for example, serve as a component type for a relation or the input and output types for an index. A survey of the problems associated with persistence and data types is to be found in [Atkinson and Buneman 85]. A more fundamental problem is what the precise definition of these types is, and how they interact with one another. In particular, one might ask whether there is a single underlying type of which each of these three types is a manifestation. Another question is how the basic operations of, say, the relational algebra act on these types. For example, is the most "natural" of all relational operations, the *natural join*, a primitive operation, is it something that the programmer can define in terms of more primitive operations., or is it something that is not directly available? In the case that it is primitive or programmer-defined, what is its type?

This paper dscribes a simple notation for partial functions, and an informal description of inheritance. A more formal description of *maps* - partial functions that are well-behaved with respect to inheritance is then derived, and it is shown how records, relations and indexes can all be derived from this one construct. It is also indicated, briefly, how this notion of a partial function connects with relational database theory. A final section outlines how type-checking can be implemented to give a somewhat more faithful representation of relational and other database operations in a typed programming language.

2. Records and Inheritance

The clue to providing a unified set of types for database comes from the idea of *inheritance*. The importance of inheritance (sometimes called *generalization* or *subsumption* has been recognized for some time in programming languages [Xerox 81], data bases [Smith 77] and in semantic networks. Recent work [Cardelli 84] has shown that, at the level of types, inheritance can be cleanly combined with functional programming; and [Aït-Kaci 84] shows that inheritance can itself serve to model computation and to provide a form of logic programming. Here we shall initially describe inheritance at the level of values, as is done, for example, in [Attardi 81]. To provide an informal introduction some notation is introduced for describing partial functions on a finite set of values. The expression,

$$\{'Susan' \Rightarrow 3490;' Peter' \Rightarrow 7731;' Karen' \Rightarrow 8535\}$$

describes a very small telephone directory - a partial function from strings to integers. If the range of the partial function contains just one element, , we may use an abbreviated description

$$\{7; 123; 22\}$$

for

$$\{7 \Rightarrow \{\}; 123 \Rightarrow \{\}; 22 \Rightarrow \{\}\}$$

Such partial functions apparently correspond to subsets of the input set of values, but we shall shortly modify our definition of partial function so that this correspondence is not entirely correct. When the input to the partial function is a set of labels as in

$$\{Idno \Rightarrow 12345; Name \Rightarrow' Jones'; BirthYear \Rightarrow 1957\}$$

we may think of the partial function as a record whose output is taken from some heterogeneous space of values.

Now there is a natural ordering on partial functions. If we think of records as descriptions of some "real world" object or or event, this ordering corresponds to the notion of "better description". For example

$$\{IdNo \Rightarrow 1234; Name \Rightarrow' Jones'; BirthYear \Rightarrow 1957\}$$

is a better description than

$$\{IdNo \Rightarrow 1234; Name \Rightarrow' Jones'\}$$

by virtue of having more fields and agreeing on those foelds that are common to the two descriptions. Similarly

$$\{IdNo \Rightarrow 12345; Address \Rightarrow \{City \Rightarrow' Philadelphia; Zip \Rightarrow 19118\}\}$$

is a better description than

$$\{IdNo \Rightarrow 12345; Address \Rightarrow \{City \Rightarrow' Philadelphia'\}\}$$

because the field values in one are themselves decriptions and are better defined in one than the other.

This last expression has introduced the possibility that partial functions may be "higer order" and that the input and output functions may be spaces on which there is an ordering. The question now arises whether any partial function we can write down in this notation is allowable. Consider the following three expressions.

$$\{\{Emp\# \Rightarrow 1234\} \Rightarrow \{Name \Rightarrow' J.Brown'; Office \Rightarrow Philadelphia\};$$
$$\{Emp\# \Rightarrow 1234; ShoeSize \Rightarrow 10\} \Rightarrow \{Name \Rightarrow' K.Smith'\}\} \qquad (a)$$

$$\{\{Stud\# \Rightarrow 3456\} \Rightarrow \{Name \Rightarrow' D.Dare'\};$$
$$\{Course\# \Rightarrow' CIS123'\} \Rightarrow \{CName \Rightarrow' DatabaseSystems'\}$$
$$\{Stud\# \Rightarrow 3456; Course\# \Rightarrow' CIS123\} \Rightarrow \{Name \Rightarrow' D.Dare';$$
$$CName \Rightarrow' DatabaseSystems';$$
$$Grade \Rightarrow' A'\} \qquad (b)$$

$$\{\{Emp\# \Rightarrow 1234\} \Rightarrow \{Name \Rightarrow' J.Brown'; Office \Rightarrow Philadelphia\}$$
$$\{Emp\# \Rightarrow 1234; ShoeSize \Rightarrow 10\} \Rightarrow \{Name \Rightarrow' J.Brown'; Office \Rightarrow Philadelphia\}$$
$$\{Emp\# \Rightarrow 1234\} \Rightarrow \{Name \Rightarrow' J.Brown'\}\} \qquad (c)$$

Example (a) is badly behaved. In return for a better input it has produced a less informative - and contradictory - output. Example (b) is the sort of behaviour one might expect from a database system. There is a extra information to be gained by providing a better specified input. Example (c) is redundant in that we can infer the second and third input-output pairs from the first, but we can nevertheless consider these pairs as part of the partial function. Of course, it is difficult to imagine how a typed programming language would allow us to assign a type to any of these examples, although several database management systems allow us to represent data like those of example (b).

Example (c) poses a question of representation. Should we think of the "redundant" part of the partial function as not being part of the partial function at all, or should we adopt the notion that a partial function is really a (binary) relation that includes all the redundant parts. We adopt the latter notion

3. A Domain Theoretic Description of Partial Functions

In order to prevent badly behaved partial functions such as the one we have just seen, we need to set up some formal apparatus to describe what partial functions are admissible with resepect to inheritance. The mathematical results in this section are all well-known in domain theory; only their interpretation with respect to data bases is new. We shall define a partial function as a subset F of $V_1 \times V_2$ where V_1 and V_2 are domains. The conditions we impose are

1. $(\perp_1, \perp_2) \in F$.
2. $(x, y) \in F$, $x' \sqsupseteq x$, and $y \sqsupseteq y'$ imply $(x', y') \in F$.
3. $(x, y_1) \in F$ and $(x, y_2) \in F$ imply $(x, y_1 \sqcup y_2) \in F$.

The second of these condiditions guarantees that the partial function is well-behaved with respect to inheritance; the third ensures that the relation F is "functional", namely that there is a unique "best" output for a given input. The first condition is for convenience in simplifying further definitions. I shall use the term *map* to describe a subset of $V_1 \times V_2$ with these properties and use $V_1 \mapsto V_2$ to denote the set of such maps. Programming language semanticists will recognise the definition of a map to be similar to the definition of an approximable mapping in [Scott 82], and this connection deserves much closer examination. However, our purpose in this paper is to examine the consequences of regarding a map as a practical data constructor.

Maps are ordered by inclusion. If F_1 and F_2 are maps in $V_1 \mapsto V_2$, we can define

$$F_1 \sqcap F_2 = F_1 \cap F_2$$

and

$$F_1 \sqcup F_2 = \cup\{F | F \in V_1 \mapsto V_2, F \supseteq F_1 \text{and} F \supseteq F_2\}$$

the latter only being defined when the intersection is non-empty. Thus maps themselves form a domain.

We can call a map *elementary* if it is generated by a single pair of elements, i.e. it is of the form

$$\{z, \perp_2 | z \in V_1\} \cup \{(x', y') | x' \in V_1, y' \in V_2, x' \sqsupseteq x, y \sqsupseteq y'\}$$

for any $x \in V_1$ and $y \in V_2$. A map is defined to be *finite* if it is of the form $F_1 \sqcup F_2 \sqcup \ldots \sqcup F_n$ where $F_1, F_2, \ldots, F_n$ are elementary. The notation of the previous section provides us with a method for describing finite maps (provided the notation is consistent with a finite map.) It follows from these definitions that if F_1 and F_2 are finite then so are $F_1 \sqcup F_2$ and $F_1 \sqcap F_2$ whenever these exist. Another useful observation is that the composition of two maps,

$$F \circ G = \{(x, z) | \exists y.(x, y) \in F, (y, z) \in G\}$$

is a map and that $F \circ G$ is finite if F is finite or if G is finite.

Having covered this groundwork, let us look at some simple classes of maps.

1. Suppose V is a flat domain of values (e.g. the integers) and $TRIV$ is the trivial domain containing one (non-bottom) element $\{\}$. The maps in $V \mapsto TRIV$ correspond to subsets of $V \setminus \{\bot\}$ with $\sqcap$ and $\sqcup$ rescpectively corresponding to set intersection and set union. The ordering is the containment ordering.
2. Suppose Λ is a finite flat domain of labels and V is any domain. The maps in $\Lambda \mapsto V$ are the *records* over V with the ordering described informally in the previous section. We shall call this domain $\mathcal{R}(V)$
3. Now consider the finite maps in $\mathcal{R}(V) \mapsto TRIV$. It is claimed that we can identify these with the *relations* over V with $\sqcup$ denoting the *natural join*.

The last example requires some elaboration. In the first place, since $\mathcal{R}(V)$ is not a flat domain there is a distinction between relations and sets (as defined in case 1 above). In fact, the members of $\mathcal{R}(V) \mapsto TRIV$ are in $1-1$ correspondence with the upward closed subsets, or *filters* of $\mathcal{R}(V)$. Secondly, in data base programming languages, we usually think of relations as a set of uniformly typed, flat, records (i.e. records whose values lie in a flat domain.) However there is no need for this in defining the natural join. For example, the natural join of

$$\begin{array}{llll}
\{\{Name \Rightarrow \text{'J. Doe'}; & Dpt \Rightarrow \text{'Sales'}; & Add \Rightarrow \{Cty \Rightarrow \text{'Moose'} & \}\}; \\
\{Name \Rightarrow \text{'M. Mac'}; & Dpt \Rightarrow \text{'Manuf'} & & \}; \\
\{Name \Rightarrow \text{'N. Bug'}; & & Add \Rightarrow \{ \quad State \Rightarrow \text{'Mo'} & \}\}\}
\end{array}$$

and

$$\begin{array}{llll}
\{\{Dpt \Rightarrow \text{'Sales'}; & Add \Rightarrow \{ & State \Rightarrow \text{'Wy'} & \}\}; \\
\{Dpt \Rightarrow \text{'Rsrch'}; & Add \Rightarrow \{Cty \Rightarrow \text{'Billings'} & & \}\}; \\
\{Dpt \Rightarrow \text{'Manuf'}; & Add \Rightarrow \{ & State \Rightarrow \text{'Mo'} & \}\}\}
\end{array}$$

is

$$\begin{array}{llll}
\{\{Name \Rightarrow \text{'J. Doe'}; & Dpt \Rightarrow \text{'Sales'}; & Add \Rightarrow \{Cty \Rightarrow \text{'Moose'}; & State \Rightarrow \text{'Wy'}\}\}; \\
\{Name \Rightarrow \text{'M. Mac'}; & Dpt \Rightarrow \text{'Manuf'}; & Add \Rightarrow \{ & State \Rightarrow \text{'Mo'}\}\}; \\
\{Name \Rightarrow \text{'N. Bug'}; & Dpt \Rightarrow \text{'Manuf'}; & Add \Rightarrow \{ & State \Rightarrow \text{'Mo'}\}\}; \\
\{Name \Rightarrow \text{'N. Bug'}; & Dpt \Rightarrow \text{'Rsrch'}; & Add \Rightarrow \{Cty \Rightarrow \text{Billings}; & State \Rightarrow \text{'Mo'}\}\}\}
\end{array}$$

Thus the definition of natural join extends naturally to "ragged" and "non-flat" (non first-normal-form) relations. In the case that $R1$ and $R2$ are each uniformly typed, our definition agrees with the usual definition of natural join, and if $R1$ and $R2$ have the same type the natural join, as expected, defines the intersection, and $R1 \sqcap R2$ gives the union.

We can take this domain-theoretic approach to relations further, and from a relation R defined on $\mathcal{R}(V) \mapsto TRIV$, define a map R' in $\mathcal{R}(V) \mapsto \mathcal{R}(V)$ by

$$(x,y) \in R' \quad \text{iff} \quad \forall r.(r,\{\}) \in R \text{ and } r \sqsupseteq x \text{ imply } r \sqsupseteq y$$

R' belongs to the special class of maps known as *closures*. One way of characterising a closure is by the following rules. A map F is a closure if

1. $x \sqsubseteq y$ implies $(x,y) \in F$
2. $(x,y) \in F$ implies $(x \sqcup w, y \sqcup w) \in F$
3. $(x,y) \in F$ and $(y,z) \in F$ imply $(x,z) \in F$

The reason for casting the closure property in this form is that any database theorist should immediately recognise this definition as Armstrong's axioms for *functional dependencies* (replace $\sqsubseteq$ by $\subseteq$ and $\sqcup$ by $\cup$ on the set of labels.) The precise connection is beyond the scope of this paper, however it is worth mentioning that we can derive several of the basic results in relational database theory by this domain-theoretic approach and extend them to ragged, non-flat relations. It should also be pointed out that the derivation of closure given above is *not* the way one actually obtains Armstrong's axioms. Moreover, for various technical reasons it is necessary to take the elements of a relation R, the set $\{r_1, r_2, \ldots, r_n\}$ such that R is the union of elementary maps generated by $(r_i, \{\})$ $(1 \leq i \leq n)$, as an *inconsistent* set, i.e. a set of elements which is pairwise inconsistent (any pair does not have a defined $\sqcup$.)

4. Implications for Data Base Type Systems

The previous section provided some theoretical justification for using maps as a basic data constructor for data base programming languages since we could use it to represent (and perhaps implement) record, relation and index types. The purpose of this section is to suggest how such structures might be practically incorporated into a typed programming language. However, we apparently achieved this uniformity by treating labels as *values* and this may give us rather more than we had bargained for. For example it is difficult to attach a meaning to maps such as

$$\{Name \Rightarrow \text{'Jones'}; 4 \Rightarrow \text{'emu'}; 17 \Rightarrow Age\}$$

since they appear to be ill typed (although they are perfectly reasonable expressions.) What kind of type system will exclude such expressions? Fortunately [Cardelli 84] has provided us with most of the groundwork, and his type system for Amber with minor modifications will support precisely the kinds of maps in whic we are interested. What follows is a very brief account of the modifications to this system that will support map types. The reader is referred to [Cardelli 84] for further details.

To start with, we remove record types from the domain of values, but extend the domain to include maps and labels[2]; thus we express the domain of values as

[2] variant records or sums are not considered here.

$$\mathbf{V} = \mathbf{B} + \mathbf{L} + \mathbf{M} + \mathbf{F} + \ldots$$
$$\mathbf{M} = \mathbf{V} \mapsto \mathbf{V}$$
$$\mathbf{F} = \mathbf{V} \to \mathbf{V}$$
$$\ldots$$

where $\mathbf{B}$ is a flat domain (or sum of flat domains) of basic values (such as integers, strings, booleans), $\mathbf{L}$ is a flat domain of labels, $\mathbf{M}$ is the domain of finite maps, $\mathbf{F}$ is the domain of functions etc.

Now there is a slight problem in doing this. In order to define an ordering on maps, we needed (at some level) to have equality defined on the domain of values $\mathbf{V}$, and equality is not decidable when $\mathbf{F}$ contains elements corresponding to arbitrary expressions in the λ-calculus. However, higher-order languages (languages in which functions are values) such as ML [Milner 83] and PS-algol [Atkinson *et al* 81] get round this problem either by having equality that *fails* on functions, or by using some retricted notion of equality such as textual equality or referential equality. We shall assume this has been done.

We now need to assign types to labels and maps. Since labels are values, we must severely restrict the kinds of "run-time" computation we can do if we want to preserve the kinds of type-checking common in languages with records. To do this we assign each label its own type. Moreover, since context should make clear whether we are referring to values or types, we shall use the same term to denote both the value and the type. Thus the type of the label $IdNo$ is $IdNo$. We now want to have rules that will type an expression such as

$$\{Idno \Rightarrow 12345; Name \Rightarrow' Jones'; BirthYear \Rightarrow 1957\}$$

as

$$\{Idno \Rightarrow int; Name \Rightarrow string; BirthYear \Rightarrow int\}$$

We also want to allow other kinds of maps in which the domain and range are uniformly types. To do this we add a type expression $\tau_1 \mapsto \tau_2$ where τ_1 and τ_2 are type expressions. Thus the type of

$$\{'Susan' \Rightarrow 3490;' Peter' \Rightarrow 7731;' Karen' \Rightarrow 8535\}$$

should be $string \mapsto int$.

To do this we use the following typing rules for maps by

1. if $e_1 : \tau, e_2 : \tau, \ldots, e_n : \tau$ and $e'_1 : \tau', e'_2 : \tau', \ldots, e'_n : \tau'$ then
   $\{e_1 \Rightarrow e'_1; e_2 \Rightarrow e'_2; \ldots; e_n \Rightarrow e'_n\} : \tau \mapsto tau'$

2. if $l_1, l_2, \ldots' l_n$ are labels and $e'_1 : \tau'_1, e'_2 : \tau'_2, \ldots, e'_n : \tau'_n$
   then $\{l_1 \Rightarrow e'_1; l_2 \Rightarrow e'_2; \ldots; l_n \Rightarrow e'_n\} : \{l_1 \Rightarrow \tau'_1; l_2 \Rightarrow \tau'_2; \ldots; l_n \Rightarrow \tau'_n\}$

3. if $e : \sigma \mapsto \tau$ and $f : \sigma' \mapsto \tau'$
   then $(e \sqcap f) : (\sigma \sqcup \sigma') \mapsto (\tau \sqcap \tau')$
   and $(e \sqcup f) : (\sigma \sqcap \sigma') \mapsto (\tau \sqcup \tau')$
   provided all meets and joins are defined.

Of these rules (1) follows Cardelli's type rules for records, (3) his rules for functions, which behave very much like functions. Only (2) is at all new.

Some examples may clarify the use of these rules. In the first place, none of these rules (and hopefully no others) will assign a type to expressions like

$$\{Name \Rightarrow 'Jones'; 4 \Rightarrow 'emu'; 17 \Rightarrow Age\}$$

Second, we can examine the type of natural join and other relational operations. Consider the declarations

type $Person = \{IdNo \Rightarrow int; Name \Rightarrow String; DeptNo \Rightarrow int\}$
type $Department = \{DeptNo \Rightarrow int; DName \Rightarrow string\};$

val $PersonRel : Person \mapsto \{\} = \ldots;$
val $DepartmentRel : Department \mapsto \{\} = \ldots;$

that describe the relations $PersonRel$ and $DepartmentRel$.

The natural join $PersonRel \sqcap DepartmentRel$ of these two relations has type, inferred by rule 3, $(Person \sqcup Department) \mapsto \{\}$ or

$$\{IdNo \Rightarrow int; Name \Rightarrow string; DeptNo \Rightarrow int; Dname \Rightarrow string\} \mapsto \{\}$$

which accords with our intuition for the type of a natural join. Note that since $\sqcup$ is not always defined, the inference rules may fail to find a type for the natural join. This would happen if, for example, we rewrote the type of $Department$ as $\{DeptNo \Rightarrow string; DName \Rightarrow string\}$.

Third the rules do assign a type to expressions such as

$$\{EmpNo \Rightarrow IdNo; LastName \Rightarrow Name\},$$

a map in which both the input and output are labels. Such maps are extremely useful in database work for relabelling records. Relabelling of a record is nothing more than composition of maps, thus

$$\{EmpNo \Rightarrow IdNo; LastName \Rightarrow Name\} \circ \{IdNo \Rightarrow 12345; Name \Rightarrow' Jones'\}$$

evaluates to

$$\{EmpNo \Rightarrow 12345; LastName \Rightarrow' Jones'\}$$

and is again well-typed. Note that $\{EmpNo \Rightarrow IdNo; LastName \Rightarrow Name\}$ denotes a type as well as a value. A special case of relabelling is *projection*, which is extended to relations in the relational algebra. For example

$$\{IdNo \Rightarrow IdNo\} \circ \{IdNo \Rightarrow 12345; Name \Rightarrow' Jones'\}$$

"projects" onto $IdNo$ to produce $\{IdNo \Rightarrow 12345\}$.

The idea that labels are types in the sense described is somewhat contentious, but may be needed if we are to write suitably generic functions for database work. Consider the problem of writing a *transitive closure* function for a relation. There is no difficulty in expresssing the transitive closure in Prolog or any other language that treats parameters positionally, but what can we do using the relational operators of a database programming language? Specifically, suppose we are given a relation of type $\{c1 \Rightarrow int; c2 \Rightarrow int\} \mapsto \{\}$ and we want the output to be a relation of the same type. This means that we have to do successive joins of a relation in which we identify the $c1$ label (column) of one relation with the $c2$ field in the other. Thus the join is not a natural join. In most programming languages that attempt to represent relations, this has to be done by unpacking the relations, forming the identification among records, and reconstructing the result from tuples. Here is a function that performs the task using natural joins and relabelling.

$$\begin{array}{l}\textbf{val } TC(R : \{c1 \Rightarrow int; c2 \Rightarrow int\} \mapsto \{\} = TC1(R, NUL, R)\\ \textbf{where}\\ \textbf{val rec } TC1(R, S, T : \{c1 \Rightarrow int; c2 \Rightarrow int\} \mapsto \{\}) =\\ \quad \textbf{if } T \sqsubseteq S \textbf{ then } S\\ \quad \textbf{else } TC1(R, S \sqcup T, (\{c1 \Rightarrow c1; l \Rightarrow c2\} \circ \circ R) \sqcap (\{l \Rightarrow c1; c2 \Rightarrow c2\} \circ \circ R)\end{array}$$

In order to perform the relabelling for the join, we have introduced a new label l. NUL is the empty relation and $\circ\circ$ is the extension of relabelling for a relation. The point of introducing this example is that we would like to be able to write a *generic* transitive closure function. To do this some form of label parameterization is essential.

5. Further Research

We have shown that taking maps as data constructors and treating labels as a special class of values provides a uniform treatment of data tyes for data base programming. What we have yet to do is to present an adequate set of operators for maps. We have seen that meet ($\sqcap$), join ($\sqcup$) and composition ($\circ$) are useful and allow us to produce most of the operations of the relational algebra. The adequacy of these operations depens, of course, on what is available in the rest of the language. Data base programming langugaes usually allow some form of iteration over relations (or some similar data type) and one can therefore implement any operation by decomposing and reconstructing relations a record at a time. The question of what is adequate therefore is not one of computational power, but what is most convenient. This is something that can only be answered by substantial experiments with various combinations of operators.

Another omission is a discussion of implementation. If we use type constraints, as has been suggested, to ensure that all maps are uniformly typed, then a B-tree or hash-table mechanism is sufficicent. In fact it is interesting to note that PS-algol uses precisely the same mechanism to resolve record labels at compile time as it does to implement run-time tables. However, were we to extend our hypothetical language to deal with ragged relations (relations with nulls), we would be faced with a more complicated indeing problem. In this connection some of the techniques suggested for implementing universal relations might be appropriate.

A more difficult problem though is to examine type parameterized programming. Languages such as Russell [Demers and Donahue 80] and Poly allow types to be treated as objects so that functions can be constructed that take types as parameters and produce new types as results. We have suggested that labels are values with individual associated types. Thus there is a special class of labet types and one might use this in a type- and label-parameterised version of the transitive closure operation shown above. Perhaps the most interesting prospect of treating labels as parameters is that data base schemas and semantic networks are large labelled graphs. A data base management systems takes such a graph, checks it for consistency, and produces what is in effect an abstract data type. It may be that this approach to data types will allow us to treat data base systems as large parameterized data types and remove once and for all the lacuna between data base systems and programming languages.

6. REFERENCES

[Aït-Kaci 85] Aïit-Kaci, H. *A Model of Computation Based on Calculus of Type Subsumption.* PhD thesis, University of Pennsylvania, 1984.

[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.

[Atkinson *et al.* 81] Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. PA-algol: an Algol with a Persistent Heap. *ACM SIGPLAN Notices* 17 (7), July, 1981. Also available as Technical Report CSR-94-81, Edinburgh University comuter Science Department.

[Atkinson and Buneman 85] Atkinson, M. P. and Buneman O. P. *Data Base Programming Language Design.* Technical Report, University of Glasgow Department of Computer Science, Glasgow, Scotland, 1985.

[Attardi 81] Attardi, G. and Simi, M. *Semantics of inheritance and ittributions in the description system Omega* Technical Report A. I. Memo 642, MIT, August, 81.

[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.

[Cardelli 84] Cardelli, L. A semantics of Multiple Inheritance. In G. Kahn, D. B. MacQueen, G. Plotkin (editors), *Semantics of Data types.* 1984 Springer LNCS 173.

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

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

[Demers and Donahue 80] Demers, A. and Donahue, J. *The Semantics of Russell: An Excercise in Abstract Data Types.* Technical Report, Computer Science Department, Cornell University, 1980.

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

[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.

[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.

[Scott 82] Scott, D. Domains for Denotational Semantics. In *ICALP 1982, Aarhus, Denmark.* July, 1982.

[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.

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

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

[Xerox 81] The Xerox Learning Research Group. The Smalltalk-80 system. *Byte* 6:36-48, August, 1981.

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.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

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

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

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

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 University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

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 EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

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

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

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

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 EUCS Departmental Report CSR-94-81.

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, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

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

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

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. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

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.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

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", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-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, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

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, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 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).

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, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Internal Reports

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

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

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.

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.

In Preparation

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 PPRR-17-85.

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 - see PPRR-14-86.

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 17th March 1986.

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.

PPRR-1-83	The Persistent Object Management System - Atkinson, M.P., Chisholm, K.J. and Cockshott, W.P.	[Printed]	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	[Printed]	£2.00
PPRR-3-83	The PS-algol implementor's guide	[Withdrawn]	
PPRR-4-83	The PS-algol reference manual - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	[Printed]	£2.00
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	[Printed]	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	[Printed]	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	[Printed]	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	[Printed]	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	[Printed]	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	[Printed]	£1.00

PPRR-11-85 PS-algol Abstract Machine Manual [Printed] £1.00

PPRR-12-85 PS-algol Reference Manual - second edition [Printed] £2.00

PPRR-13-85 CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P. [Printed] £2.00

PPRR-14-86 An Integrated Graphics Programming Environment - second edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P. [Printed] £1.00

PPRR-15-85 The Persistent Store as an Enabling Technology for Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P. [Printed] £1.00

PPRR-16-85 Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R. [Printed] £15.00

PPRR-17-85 Database Programming Language Design - Atkinson, M.P. and Buneman, O.P. [Printed] £3.00

PPRR-18-85 The Persistent Store Machine - Cockshott, W.P. [Printed] £2.00

PPRR-19-85 Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R. [Printed] £1.00

PPRR-20-85 Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P. [Printed] £1.00

PPRR-21-85 A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D. [Printed] £1.00

PPRR-22-86 Some Applications Programmed in a Persistent Language - Cooper, R.L., Cranston, R.D., Dearle, A. and MacFarlane, D.K. [In Preparation]

PPRR-23-86 PS-algol Applications Programming - Cooper, R.L., Dearle, A., MacFarlane, D.K. and Philbrow, P. [In Preparation]

PPRR-24-86 A Compilation Technique for a Block Retention Language - Cockshott, W.P.and Davie, A.J.T. [In Preparation]

PPRR-25-86 Thoughts on Concurrency - Wai, F. [In Preparation]

PPRR-26-86 An Exception Handling Model in a Persistent Programming Language - Philbrow, P. [In Preparation]

PPRR-27-86 Concurrency in Persistent Programming Languages - Krablin, G.K. [In Preparation]

PPRR-28-86 A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. [Printed] £1.00

PPRR-29-86 Extracting Garbage and Statistics from a Persistent Store - Campin, J. [In Preparation]

PPRR-30-86 Data Types for Data Base Programming - Buneman, O.P. [Printed] £1.00