IPC Technical Report 92-008

Programming over a Persistent Data Space

John L. Pfaltz

IPC-92-08
August 19, 1992

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract:

The purpose of this report is to describe semantic properties of languages that support direct access to items in persistent data spaces. These languages differ in non-trivial ways from more familiar languages which describe processes operating over data represented in a transient memory.

These semantics are presented with respect to a formal Turing machine model, which we first develop. Then, we examine issues associated with type and class inheritance which we regard as distinct concepts, with symbolic naming, with set operation semantics which are somewhat surprising, and with data deletion which is quite difficult. Finally, we hint at a few of the differences in programming psychology which are engendered by such semantics.

This research was supported in part by DOE Grant #DE-FG05-88ER25063. and by JPL Contract #957721.

Our goal is to examine necessary characteristics of programming languages that allow a programmer to directly reference and manipulate persistent data. It would be linguistically convenient if persistent data could be treated in precisely the same manner as a program's non-persistent data. Various authors have proposed this as a language design goal, e.g. [AtB87]. In this report, however, we will argue precisely the opposite; that programming languages must necessarily distinguish between persistent and non-persistent data; and that familiar data types associated with non-persistent data in an executing program need not be appropriate for persistent data.

Our first task will be to characterize just what we mean by *persistent data*. One frequently encounters definitions of persistence in terms of a persistent recording medium, or in terms of its long life. While we intuitively think of persistent data as residing in files on disk or tape storage, rather than RAM, the introduction of memory resident databases [SaG90, Son89] clearly forces a re-evaluation of such an intuitive definition. Persistent data is often long lived, but it need not be. And long lived data need not fit our sense of persistent data; the local variables of a non-terminating process such as an airline reservation system can have a very long life. The real essence of persistence seems to be that it exists independent of any particular process, that multiple processes can access it[1], and that, unlike local variables in a stack model, the data can be changed only by a deliberate rewriting of it.

To discuss programming over a persistent data space we will need a semantic model of such computation. For this we use a Turing machine model, and in the next section develop a number

[1] While persistent data must be capable of being shared by several processes, this property by itself is not sufficient to characterize persistence. Data may also be shared between processes by a variety of argument passing mechanisms in procedure calls, or by explicit message passing, e.g. [Hoa78]; none of these need imply persistence.

of fundamental concepts based on it. Although it is not our purpose, this development can also serve to describe the semantics of object-oriented databases as well. Then, in Section 2, we will develop a distinction between data types and element classes that appears to be fundamental in the development of distributed heterogeneous databases. And, we will exhibit two rather surprising results concerning the semantic behavior of union and intersection operators on sets of data. In Section 3, we will address the issue of expanding the name space of a programming language to accommodate the naming of persistent data, and finally, in Section 4, we examine issues associated with the deletion of persistent data.

1. Basic Concepts

For the purposes of this report, a datum is a bit string of finite, but arbitrary, length and data is a countable collection of such bit strings. Then, in terms of a Turing machine model of computation, a *data space*, denoted by DS, is a tape on which individual datum, separated by one or more blanks, have been enumerated by a Turing machine, E. We shall refer to these bit strings which constitute the elements of the data space, DS, as *data elements*, or just *elements*.

In addition to the bits comprising a data element, each element can also be identified by its ordinal position in the enumeration, that is the 1^{st}, 2^{nd}, 3^{rd}, ..., n^{th}, ..., and so on. We call this ordinal position a *unique element identifier*, or more simply just a *unique identifier*, denoted *uid*.[2] This leads to the well-known, fundamental implication of data management:

$$uid_1 = uid_2 \text{ implies } uid_1.bits = uid_2.bits, \tag{1.1}$$

that is, if two unique identifiers are identical they must denote the same datum, and therefore the same bit string. Here we are using the functional suffix *bits* and dot notation to emphasize the

[2] Alternatively, we could call the ordinal position of an element, its *location*, since it describes its location in the enumeration — or possibly its *address*. Either would suggest the approach, common in all programming languages, by which data is identified by its storage location. However, both location and address carry connotations of fixed length word sizes or disk blocks associated with a particular implementational architecture which we would like to avoid for now.

bits comprising the element, in contrast to its *uid* denoting its position. Thus *uid.bits* and *uid* must be regarded as being of a fundamentally different nature.[3] In programming languages which support pointer types the latter is called a *pointer* which denotes a datum; the former is treated as a data *value* which may be typed in the manner of [CaW85]. Given a pointer value, or *uid*, in C or Pascal for example, one must functionally dereference it, as in `*uid` or `uid`$\uparrow$ to denote the referenced bit string.

Readily, the converse need not be true; the same sequence of bits could occur repeatedly in the enumeration. However, the contrapositive

$$uid_1.bits \neq uid_2.bits \quad \text{implies} \quad uid_1 \neq uid_2$$

must, of course, hold.

Assuming the converse is true, leads to the relational model of data, in which any two data elements (tuples) must be distinct as bit strings. With this assumption, a data element can always be uniquely identified by its constituent bits (or more often by some subsequence called its *key*) so that the concept of an *uid* is logically unnecessary. If the reader prefers the more restrictive relational model, he can replace uid_i with key_i in much of what follows. The implications above are then simply those associated with functional dependence [Mai83].

The fundamental assumption of computation is that every process can be modeled by a Turing machine [HoU79]. We will use the symbolism P_i to denote processes, or their equivalent Turing machines. Moreover, since we wish to examine multiple processes that operate on common data, perhaps concurrently, we will assume that all such processes/Turing machines can read and write this common *data tape*, DS. Each may have additional tapes to represent local memory.

---

[3] Both Beeri [Bee90] and Hull [HWW91] have made this observation, although the latter then asserts that "the distinguishing feature of [*uid*]'s is that they *uniquely identify* objects from the real world — the objects being identified", while Beeri dismisses them as "an implementation concept". In our model, a *uid* only identifies a bit string. And since a program is an implementation of process, and since a programming language is a description of such implementations, they become central to this report.

Concurrent processes must execute in time, and for our purposes it is convenient to regard time, denoted t_k, as discrete and to assume that no two operations occur simultaneously. The basic capabilities of a Turing machine is to read and write on its tapes. We extend this to include the two atomic operators to manipulate the common tape DS,

$$ds\_read\ (uid_i, P_j, t_k)$$
$$ds\_write\ (uid_i, P_j, t_k, bits)$$

where these are interpreted as: process P_j reads/writes uid_i at time t_k. Note, that the *bits* that overwrite those of uid_i in a write operation need not be of the same length as the existing datum. If the new datum is shorter, there is no problem; the remaining bits can be overwritten with blanks. If the new datum is longer, we may assume that all data preceding uid_i are shifted left a sufficient number of spaces; since regarding the tape as 2-way infinite does not change the power of the model. If we wish to refer to either DS operation, without regard to reading or writing, we will simply use the notation $ds\_op\ (uid_i, P_j, t_k)$.

We do not allow a process to insert a new datum in the middle of the enumeration, nor delete an item from the enumeration. We assume that the data space DS is initially created by enumeration, and that it can be enlarged only by further enumeration using

$$ds\_enum\ (uid_i, P_j, t_k, bits)$$

where uid_i is returned by *ds_enum* to denote the ordinal position of *bits* in DS.[4] However, a Turing machine process, P_j, may be enumerating additional data concurrently with the reading and changing of existing data by it, or other Turing machine processes, P_k. The conceptual time that DS element uid_i was enumerated is denoted t_k. One can regard t_k in all these operators as a time stamp that is returned on completion of the operation.

[4] In practice, one employs a *uid* server to keep track of the number of elements in DS and return the next consecutive *uid*.

The assumption that all DS operations are atomic; occur at distinct times[5]; and that two or more processes may read/write the same DS element can be more formally expressed as: if *ds_op* (uid_i, P_j, t_1) and *ds_op* (uid_i, P_k, t_2) are any two data space operations, then $t_1 \neq t_2$. P_j and P_k need not be distinct processes. The property of Turing machine enumeration by which we assume the data tape is created can be formally expressed as: if *ds_enum* (uid_{i_1}, P_j, t_1, *bits*) and *ds_enum* (uid_{i_2}, P_k, t_2, *bits*) are data enumeration operations, then $t_1 \neq t_2$ and $uid_{i_1} \neq uid_{i_2}$.

Let

p_initiate (P_j, t_{k_1})
p_terminate (P_j, t_{k_2})

be operators which initiate and terminate a Turing process P_j at t_{k_1} and t_{k_2} respectively. Readily, for any process P_j, we must have $t_{k_1} < t_{k_2}$. And if uid_i denotes a DS element where *ds_enum* (uid_i, P_j, t_1, *bits*), and if P_k is a process where *p_initiate* (P_k, t_2) and *p_terminate* (P_k, t_3), then any DS read/write operation *ds_op* (uid_i, P_k, t_4) will be meaningless unless, $t_2 < t_4 < t_3$ and $t_1 < t_4$. That is, DS operations must be issued by processes while they are executing, and no DS read/write operation can refer to a *uid* which has not already be enumerated.

1.1. The Concept of Persistence

Following this development, we would assert[6] that the first fundamental property of persistence can be expressed by the implication

---

[5] One could permit two *ds_read*'s to occur simultaneously, but it would add nothing to our development.

[6] We will make a number of assertions throughout this report. It is our intention to have them highlight important points, as well as serve as analogues to propositions in a mathematical paper. Although we give supporting arguments, only a few of these assertions can be "proven" in any formal sense. This is characteristic of the inductive sciences. Neither Boyle's Law nor the assertion "all crows are black" can be proven; but both can be disproven. Consequently, they can be used to test (the original meaning of to prove) the material of the report. They should be stated with sufficient precision that counter examples, if the exist, can be demonstrated. We urge the reader to search for such counter examples.

Assertion 1.1: If $ds\_read(uid_i, P_i, t_1).bits \neq ds\_read(uid_i, P_j, t_3).bits$ **and** $t_1 < t_3$**, then there must exist** $ds\_write(uid_i, P_k, t_2, bits)$ **where** $t_1 < t_2 < t_3$**.**

Essentially, data is persistent if the bits comprising a data element can only be changed by a process that deliberately rewrites it. If the data element can be changed by the initiation or termination of a process, or losing power, or by any other mechanism, it is not persistent.

However, the property above does not by itself characterize persistence as we intuitively understand it. For example, if $P_i = P_j = P_k$, then this property is true of all data denotational schemes, whether persistent or not. A variable in a process P_i does not change its value unless it is changed by the process. Nor is requiring P_i, P_j, and P_k to be distinct sufficient to make this property characterize our intuitive understanding of persistence; the three processes could be distinct sub-processes of a single process which has passed the identity, or uid_i, of the data element to each of them.

Assertion 1.1 really expresses the property of persistence *over the interval* $[t_1, t_3]$. To be persistent, we might require a data element uid_i to be persistent over the half open interval $[t_k, \infty)$, where $ds\_enum(uid_i, P_j, t_k, bits)$. But not only is such a concept involving ∞ difficult to use, variables in non-terminating processes also satisfy this condition. We believe that the following assertion characterizes persistent data as it is intuitively understood.

Assertion 1.2: The data element denoted by uid_i **is** *persistent***, where** $ds\_enum(uid_i, P_1, t_1, bits)$**, if it is persistent over the interval** $[t_1, t_2]$**, where** $t_2 > \max_{t_k}\{p\_terminate(P_j, t_k) \mid p\_initiate(P_j, t_{k'}), t_{k'} < t_1\}$**.**

That is, the data element uid_i persists after all processes which were initiated before uid_i was enumerated have terminated.

1.2. Identifiers, Data, and Data Names

In [CaW85], Cardelli and Wegner assume an initially untyped data space. One must approach this assumption with caution in our Turing machine interpretation. We also assume that the data elements, or bit strings, themselves are untyped. However there is a fundamental differ-

ence between a data element and its ordinal position, even though the latter may also be expressed as a finite bit string.

For implementation efficiency, it is convenient to regard pointers, or *uid*'s, as integer data types and most languages do so. However, the operations on each should not be the same. Only the equality, inequality, and successor operators are really meaningful for ordinal enumerators. The latter can be implemented by integer addition, as in the manipulation of C pointers; but clearly the other integer operators multiplication, division, and exponentiation are, or ought to be, undefined. Consequently, we claim

Assertion 1.3: The semantics associated with unique element identifiers and the element bit strings themselves should be distinct; and the syntax of any language should reflect this semantic difference.

Programming with numeric addresses, or ordinal identifiers, is at best aggravating. We quickly eschew coding in binary and introduce equivalent mnemonic element names.[7] But what does the name denote? In assembly languages, the name is a synonym for a memory location, or in our sense an identifier. In higher level languages, the variable name is also bound to a storage location, say *loc*; but denotes, in the language, its contents or *loc.bits*.

This leads to the delightful confusion in compiler theory of *l-values* and *r-values* [AhU79], or *location* and *value* [Pra84] of a variable name, say *x*, when used in an assignment

$$x \;=\; x + 5. \tag{1.2.1}$$

On the right side, the expression is evaluated as "$r\text{–}value(x) + 5$", while we want $l\text{–}value(x)$ for the left hand x in order to generate an operation of the form

$$write(l\text{–}value(x), P_j, t_k, r\text{–}value(x) + 5).$$

In this report, a persistent *element name* will always be syntactically synonymous to an identifier, or $l\text{–}value$, and assignments such as that above will be written[8]

---

[7] Even in assembly languages, the mnemonic names that replace absolute, virtual, or relocatable addresses may denote more than just data. They may denote entire procedures or statements within a code segment. For the most part, however, we are concerned only with names that denote data elements.

$$x.bits \leftarrow x.bits + 5. \qquad (1.2.2)$$

A symbolic name must, at some point, be bound to that which it denotes. Binding times may vary widely. In the case of variable names, both their *l−values* and *r−values* must be bound. The former binding may occur at compile time, if the storage location is absolute; at load time if it is relocatable; or at run time if it is virtual or a stack offset. The *r−value* binding occurs whenever an initialization or assignment is executed.

When are persistent element names bound to unique identifiers? The preceding binding times were all expressed relative to a single process P_j. Because, persistent element names can be shared by several processes, their binding must be independent of any particular process that employs them. Just as with more conventional languages, the mechanism of binding may vary according to the desires of the language designer[9]; nevertheless, we may assert

Assertion 1.4: If persistent data can be named, then a name space binding symbolic names to unique element identifiers must be maintained. Moreover, this name space, together with every name in it, must itself be persistent.

By asserting that the name space, which we will denote by NS, is itself persistent, we are asserting that multiple processes can access it; in particular, any language translation process must have access to the name space; and that element names will be synonymous to *uid*'s. One can easily envision the binding pairs (*name, uid*) of NS as elements of the same persistent tape, DS. However, it can be conceptually convenient to regard NS as a separate persistent tape. A more important programming language issue is whether these bindings should be allowed to change.

In the assignment (1.2.1), there are two different kinds of names. The name "5" is a *constant* in that the name is permanently bound to its value for the duration of its life [Pra84]. The name "*x*" is a variable name, in that the value it denotes may vary, even though the storage

[8] To have an arithmetic expression on the right side of the assignment such as this requires type information about the interpretation of *x.bits* which we have not yet developed.

[9] In a language we have implemented, we let the enumerating process, E, which adds new elements to the tape also bind its symbolic element name, if any, to the ordinal identifier. But other mechanisms seem equally viable.

location will not — so long as the name is within scope.

As we have defined them, unique identifiers always denote the same element (whose constituent bits may change) in the enumeration. Should data element names, which are linguistic synonyms, also be invariant? In many cases, the answer is yes. If an element is named, say "*adam*", then we would expect "*adam*" to denote the same element in every process that employs it. Like "5", "*adam*" then functions as a constant name, except that it is the element identity, or *uid*, which is invariant, not the element's bits.[10] By a *element name* we will mean a symbolic name in a language whose binding to a unique identifier can not be changed. It is a linguistic constant. Those symbolic names whose bindings to persistent data can be changed, typically at runtime, we will call *variable element names.* These will function very much like pointers. In this report, all element names, unless specifically prefixed as variable, will be assumed to be persistent, constant, element names.

If we assume that the meaning of persistent element names, at least in so far as they are bound to unique identifiers, is invariant, then we need to be able to distinguish constant element names from variable element names and ordinary process variable names in order to avoid assignments of the form

$$adam \leftarrow <different\ uid>. \tag{1.2.3}$$

No programming language allows its constants to be redefined, as in

$$7 \leftarrow 5+3.$$

We should perhaps note that the usual semantic interpretation of the integer 5 in the assignment above, and in expressions (1.2.1) and (1.2.2) have no counterpart in the model we have developed. The value 5 may exist as a bit string on the tape, but it is identified by its ordinal *uid*. If "5" is to be a constant name, a bit string denoting the value 5 must be enumerated on the tape,

[10] A good example of such naming can be found in [AbK89] which develops a functional model of object oriented query languages that reaches some of the same conclusions found in this report. They include a lucid example with named elements *adam*, *eve*, *cain*, *abel*, and *seth*, from which we have shamelessly borrowed.

say by a language translation process, and its enumeration *uid* bound to the symbol "5".

A persistent name space, NS, is global. As such it introduces the same advantages and liabilities as global variables in a program's name space. The most serious liability is that an element name is bound to the element identifier across all procedures. Even popular data element names, such as "*x*" or "*count*" can only be bound once; and once bound they may not be reused by any other process sharing the same name space.

Most humans have a rather limited set of mnemonic data names relative to the quantities of data to be identified. In Section 3 we will examine ways of naturally expanding a programmers name space. But, even so, explicitly naming every element in a persistent data space, in the way that the variables in the local data space of a process are named, will prove impossible. Rather, one must name entire sets of data elements. Consequently,

Assertion 1.5: Any programming language which references persistent data must provide general mechanisms for creating, naming, and operating on collections, or sets, of data elements.

This characteristic, in itself, will distinguish languages that access and manage persistent data from those that do not.

A number of assertions have been made in this section regarding the naming of persistent data. It might be wise to test them against experience. Sequential files represent the earliest, and most basic, form of persistent data. A sequential file is a set of elements, called records.[11] The file name identifies the set. The file name itself is persistent, out living any process which opens or closes the file. Sequential access provides a primitive mechanism for looping over the set. A set union operator can be implemented by a file merge process. If one extends the sequential file to provide for direct access, as in an indexed sequential file, then the record id functions in a manner that is completely analogous to a unique identifier. If symbolic access by name is

[11] In fact, it is an ordered set, not unlike the Turing machine's tape.

desired,[12] it is frequently implemented by a B-tree index [Com79], which binds the set of symbolic names to file addresses in a persistent structure. Our treatment is completely conformable to these traditional usages. But, files and their indexes are usually regarded as auxiliary to the programming language itself. Our goal is to investigate what must be involved if they are to be included as an integral part of a programming language that can access persistent data as its operands.

1.3. The Relational Model

We now briefly examine the relational database model [KoS86, Mai83]. Unlike the model of persistent data we have been developing, the pure relational model is completely value based. A *relation* is a set of bit strings (of equal length) called *tuples*. It is a mathematical set, in that the position of any tuple in the set is irrelevant, and in principle, unknowable. The fundamental assumption of the relational model is that some subsequence of bits in the tuples of a relation, called a *key*, is sufficient to uniquely distinguish the tuples of the relation. There are no tuple identifiers.

The sequence of bits comprising a tuple element is seldom, if ever, regarded as a single value. Instead, it is subdivided into distinct subsequences, called *attributes*, which individually denote data values. Such attributes are typically named, say a_1, a_2, ..., a_k.[13] Thus, if we let t denote a single tuple, as is customary in the tuple calculus, the expressions

$$t.a_1,\ t.a_2,\ ...,\ t.a_k$$

denote the corresponding bit subsequences of t in precisely the same way that we used *uid.bits* in Section 1 to denote the bits comprising any data element. The *schema* of a relation, $R = \{ a_1, a_2, \ldots, a_k \}$ can be regarded as simply an enumeration of the attributes, or bit subsequences, comprising a tuple.

---

[12] Symbolic access is more often by content rather than name.

[13] In practice, of course, attribute names are usually mnemonically chosen, such as *name*, *age*, etc.

The advantages of this relational approach are two-fold. First, it is conceptually clear. Tuples provide a clean mechanism for aggregating logically related data values as a single element in a k-ary mathematical relation, whence its name. Second, its implementation is relatively straightforward. Tuples are easily visualized as formatted records in a sequential file, in which case attributes can be interpreted as fields, or visualized as record structures in Pascal, C, or C++, in which case the attributes name structure members. This accounts for the natural extensions of Pascal to Pascal/R [Sch77] and C++ to EXODUS [RiC87, RiC89] or ODE [AgG89].

While the relational model, as originally formulated by Codd [Cod70] is purely value based, it need not be. Both the domain and tuple calculi have been shown to be equivalent to the relational algebra, e.g. Chapter 10 [Mai83]. The tuple calculus makes extensive use of tuple identifiers, as in the formula

$$\{ x \in usedon \mid (\exists y \in instock)\,[\, x.part\_nbr = y.part\_nbr \wedge y.quantity \geq 100 \,]\,\} \qquad (1.3.1)$$

which corresponds to the algebraic expression

$$usedon \bowtie \sigma_{quantity \geq 100}(instock)$$

found in that chapter. The names *usedon* and *instock* are persistent names denoting sets in the data space, and x and y are logical variables (one free and one bound), or variable tuple names that can denote any tuple in these sets. This example indicates how tuple identification can be handled within the relational model. Moreover, many implementations routinely include a "hidden" unique identifier attribute to every tuple to facilitate join operations and to ensure the existence of keys. Relational database languages based on the semantics developed here can be implemented; however care must be taken regarding the semantics of the relational operators $\cup$ and $\cap$, as discussed in the following sections.

2. Categories of Data

We have distinguished between *uid*'s and the elements they denote. Both are bit strings; the former is an ordinal number represented as a binary integer, the latter is an arbitrary bit string.

It is the most basic categorization of data. In this section we refine the categorization of those bit strings comprising data elements. We will distinguish between bit strings which are to be regarded as single entities, or values, and those which are compositions of smaller substrings. The former we will assign to "types"; the latter to "classes". An important category of data will be those data elements comprised of *uid* substrings; these we will call "sets". Finally, we will examine the implications of class hierarchies with respect to set operators.

2.1. Types and Classes

By a *data type* we mean an interpretation of a bit string. A bit string declared to be *real* will be interpreted, that is processed, differently from one declared to be *integer*. For example, in our model if one wants to "add" two bit strings, a different addition process, or Turing machine, must be invoked if the strings are to be interpreted as integer from that invoked if they are to be interpreted as real.

There are a variety of ways of introducing type information into our model semantics. In Section 1, we employed the notation *uid.bits* to denote the bit string comprising the uid^{th} datum. Instead of the generic designator *bits*, we might have used *uid.integer*, *uid.real*, or *uid.char* to denote a typed bit string — one that is to be interpreted as a integer, real, or character value, respectively. A Turing process INT_ADD, given two datum identifiers uid_1 and uid_2 may return a value regardless of corresponding bit strings, but to be semantically meaningful both arguments should admit interpretations as uid_1.*integer* and uid_2.*integer*.

The important point is that data types and their interpretations are a property of the processing environment rather than any programming language *per se*. The bit string that is interpreted to be an *integer* 5 (00...101) on a SPARC will be quite different from that on a Intel 80x86 (00000101...00).[14] The bit string that will be interpreted as the character 'a' in an ASCII

[14] This is simply an example of the byte order difference between its big and little Endian representations [HeP90].

environment (01100001) is a different bit string in an EBCIDC environment (10000001). And Fortran character strings have a different representation than C character strings in any architecture. Processes operate in an particular environment, so that the traditional language data types, *integer*, *real*, and *character* denote different interpretations depending on the language/architecture environment. If the items of the data space are to be shared by processes operating in different environments, then the bit strings of DS may have to be coerced into the form expected by the particular processing environment. This constitutes the heart of the data heterogeneity problem that has hindered the development of widely shared databases [ScY89, SeL90, TTC90].

For now, we will use *class* to mean a pattern which only describes the format of an data element; its decomposition into substrings.[15] It can be regarded as only a linguistic construct that defines those data denotational expressions which are well-formed with respect to the data space. It is independent of the processing environment. This distinction between types and classes is non-standard, and will require amplification.

Consider a relational scheme R = { a_1, a_2, ..., a_k } from Section 1.3. It defines a class. It declares that for any given tuple t belonging to the class, the attributes a_1, a_2, ..., a_k are defined on t. Or, on our Turing machine tape, that the bit string, *t.bits* can be decomposed into corresponding substrings,[16] and the expressions $t.a_1$, $t.a_2$, ..., $t.a_k$ will be valid, well-formed expressions. Each expression $t.a_i$ denotes a bit string, that is meaningful within the language. If the attribute is typed, say as *integer*, then the bit string denoted by $t.a_i$ may be interpreted as an integer. In most relational languages, the declaration of an attribute name also establishes its

---

[15] One can extend the meaning of a class to include predicates which must be true for all instances of the class [PfF92], or to methods which are defined on these instances [Str87]. For the purposes of this report they will not be needed. However, all of the assertions of this and the following section will be true for these more general extensions of the class concept.

[16] A tuple need not be represented as a single bit string, and attributes need not be substrings or fields, c.f. [PFG92]. But visualizing it this way provides a clear semantic interpretation; and it can serve as the basis of an implementation.

attribute domain, type, or interpretation. For example, we might declare the attribute a_1 to be integer, a_2 to be real, and a_3 to be strings of 10 characters. Thus the expression, $t.a_2$, not only denotes a bit string, it also conveys the appropriate interpretation (or type) for that string.

Let $t.a_i$ be a well-formed expression in a language. Its $l-value$ semantics is that of a unique element identifier as established by our model of persistent data spaces — the subject of this report. Its $r-value$ semantics, or type, is dependent on the processing environment. We have been concentrating on *integer*, *real*, and *character* types because they are familiar. If the processing environment can process bit strings of type *image*[17], then this model of a persistent data space will support reference to them in the data space, together with their access and delivery as operands to such a process.

In Section 1.2, we observed the fundamental distinction between the ordinal position (enumeration *uid*) of an element and the bit string comprising the element; and asserted (1.3) that semantics of unique identifiers and the data themselves must be distinct. For this reason, we will call any expression denoting a bit string whose interpretation is dependent on the processing environment, a *value designator*. The expressions $t.a_1$, ... , $t.a_k$ are value designators. If the expression denotes a bit string which is to be interpreted as a unique identifier, we will call it an *element designator*.[18] Data element names are element designators. The expression t by itself, denoting a tuple element, is an element designator as are all data element names. Value designators are typed. Element designators belong to a class.[19]

---

[17] Presumably, very long, possibly variable length, bit strings.

[18] We choose this terminology to suggest "a generic element of the data space". The term "data object" is frequently used, so we might call it an "object designator". But, with object-oriented programming, the term "object" has ceased to be generic and has taken on some very specific connotations which we wish to avoid.

[19] In the object-oriented model, one makes no distinction between composite objects and objects that are treated as single values, as we have. There are only "objects". However, the latter are frequently said to belong to a "primitive class", and "for performance reasons, if the domain of an attribute is a primitive class, the values of the attribute are directly represented; that is, instances of a primitive class have no identifiers associated with them" [Kim90]. That is, they are actually implemented according to our model.

With this formulation, a C array declaration such as

```
int    count[20];
```

combines both a type and a class declaration. It declares that expressions, such as `count[15]`, or `count[i]`, or `count[3*k-2*i]` are well formed value designators (provided the subscript evaluates to $0 \leq i < 20$), whose interpretation is to be integer. This interpretation is illustrated by early Fortran syntax in which arrays were declared by

```
DIMENSION  COUNT(20)
INTEGER    COUNT
```

Similarly, a Pascal record declaration, such as

```
type CELL_ptr = ↑CELL;
type     CELL = record
                   size:          real;
                   count[0..19]:  integer;
                   next:          CELL_ptr;
                end;
```

also combines both type and class information. If *x* is an element designator of class CELL, then *x.size*, *x.count*[3], and *x.next* are all well-formed expressions. The first two are value designators, the latter is an element designator, or pointer to a data element of class CELL. In our terminology, any expression of type CELL_ptr is an element designator.

Two questions should be apparent. First, if element designators are nothing more than dressed up pointers[20] — why not simply call them pointers? In a sense they are. But the term "pointer" is inextricably associated with the concept of storage locations and addresses, whether virtual or not. Persistent data can not be identified by an addressing concept. It must be identified by an immutable symbolic element identifier. These *uid's* may be bound to memory locations, virtual addresses, or disk sector addresses. But such bindings can be transient[21], the

[20] In ODE [AgG89], persistent data is accommodated by simply designating two different kinds of pointers — ordinary pointers (to objects allocated in either a heap or stack) and persistent pointers, together with dual persistent constructors and destructors. The resulting duality must then be maintained by all processes using these pointers.

uid itself can not.

Second, we may ask why distinguish between the notion of a type and that of a class? Typed languages, such as Pascal and C, have been extremely successful. Moreover, in the class concept of object oriented languages, the thrust has been just the reverse. The methods of a C++ class are explicit interpretations, in the form of code describing operations on the objects of the class. This question is harder to answer, just because it can be very convenient to syntactically combine the two concepts; nevertheless we would assert

Assertion 2.1: Programming languages that support processes in heterogeneous systems over shared persistent data should make a clear distinction between those features which are environment dependent and those which are dependent on the semantic structure of the data space.

Our distinction between types and classes is an effort to do just that. Moreover, in Section 2.3, we will demonstrate inheritance concepts that make sense for classes, but not for types.

2.2. Sets and Set Operations

In Section 1.2, we asserted that languages which deal with persistent data must have substantial set manipulation capability. Yet, sets have always been a problem for programming languages. They are a fundamental mathematical concept; their logical semantics are well understood; and yet, none of the standard programming languages provide for a generic set concept.[22] Part of the reason has been that there is confusion between sets of values[23], such as

$$\{\ 0.0 \leq x \leq 1.0\ \}$$

and sets of elements, as denoted by pointers.

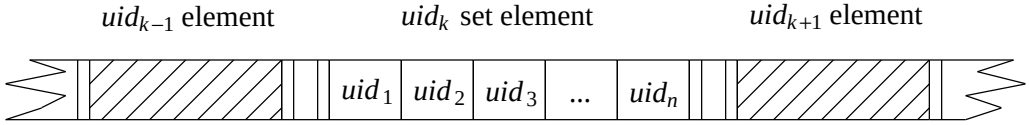
---

[21] The way datum *id*'s are bound to long term storage and to volatile memory processing is crucial to an effective implementation.

[22] Pascal sets must be of limited size. Semantically, they are really only fixed length bit strings with Boolean operators defined on them. SETL [SDD86] is a complete set-based language, but it has not gained wide acceptance. Icon [GrG83] too, may be regarded as a set language, although its generators (which are in many ways analogous to our enumerator) really generate sequences, or ordered sets.

[23] A relation is a set of values; the relational model employs value based set semantics.

For us, a set will always means a set of elements, or more properly, a set of their *uid*'s. In Turing machine tape semantics, a set is a single, possibly long but always finite, bit string comprised of enumeration *uid*'s separated by special markers. As illustrated in Figure 2-1, each set, as a single string on the DS tape, is itself an element of the persistent data space and has its own *uid*.



Representation of a Set as a single string of *uid*'s
Figure 2-1

Given the semantic concept of a set as a specified sequence of of *uid*'s, it is easy to define the standard set operators, *union*, *intersection*, and *relative complement*[24] on them, as in

```
  set1  union  set2
set1  intersect  set2
  set1  minus  set2
```

and to form set expressions such as

```
(set1  union  set2)  minus  (set3  intersect  set4).
```

Other set expressions, such as (1.3.1), or its equivalent formulation,

```
{ x in usedon | (exists y in instock)
                    [ x.part_nbr = y.part_nbr and y.quantity >= 100 ] }
```

are easily written and semantically clear. They are data element access expressions. Here, *x* is a free variable denoting any element of the set *usedon* for which the following predicate evaluates to true. It is a variable element name whose binding is changed in the course of execution.

[24] Establishing the semantics of a unary complement is more difficult. Even in mathematics, it is interpreted as the complement with respect to a universe, *U*. The question is, how does one define, or interpret, *U*? Should it be the universe of all *uid*'s, or all *uid*'s so far enumerated, or all *uid*'s denoting elements in the same class? Can the universe include elements which could be consistently included in the data space, but which have not as yet been specifically enumerated? Or should the complement of a set of bit strings be all those bit strings that are not in the set? This is the interpretation in formal language theory, which admits infinite sets. This question constitutes the heart of the issue regarding *safe queries* in relational database theory [Mai83].

Given the power to create set expressions, we would expect an assignment operator, such as

```
result ← { x in set1 union set2 | x.a1 = 17 }
```

This assignment is semantically valid, whereas the assignment (1.2.3) is not, because *result* denotes a set and we assume that the set of *uid*'s comprising the set expression on the right becomes the set of *uid*'s comprising the set denoted by the element name *result*. We are changing *result.bits*, not its associated *uid*. This then constitutes shallow copy semantics for set assignments of this form.

In addition to set assignment, we should be able to iterate over a set of elements, as in the following looping statement,

```
for_each  x  in  result  do
       value1  ←  x.a1
              .
              .
       valuek  ←  x.ak
end_do
```

In these statements and expressions we have been conceptually operating on, or assigning to, or looping over *elements* in the persistent data space, not bit strings. This is an important semantic distinction. For two distinct elements in *result*, say uid_1 and uid_2, we may have $uid_1.bits = uid_2.bits$. Herein lies the key difference between the semantics of this model of persistent data and that of the relational model of data.

Assertion 2.2: The fundamental semantic characteristic of any programming language that manipulates sets, is whether they are considered to be sets of distinct identifiers denoting data elements, or sets of distinct element bit strings.

Both semantic models are viable. Each model has its associated implementation costs which we will discuss more fully in the following section.

Nothing in our treatment of persistent data elements in DS so far has required a class concept. However, the introduction of sets changes that. Regardless of one's semantic model of sets, it is clear that one must be able to distinguish between a set and the elements comprising it. There must be a conceptual class, say SET, to which all data elements that can be semantically manipulated as a set must belong. This being so, all other data elements automatically belong to

the class "not-set", or more simply just a generic CLASS.

What we have done is to create a very coarse equivalence relation on, or partition of, DS into just two equivalence classes, from whence the term "class" arises. Data elements in the same class are semantically equivalent; e.g. they support the same data denotational expressions. In most data dependent applications it is convenient to provide a much finer partition than just these two gross equivalence classes. For instance, in preceding sections we introduced the tuple schema R = $\{ a_1, a_2, ..., a_k \}$ and record structure CELL = $\{ size, count[0..19], next \}$ as representative classes. Both are refinements of the generic CLASS. Both are equivalences, in that any two elements uid_1, uid_2 of R (or uid_3, uid_4 of CELL) support identically the same data denotational capacities, such as $uid_1.a_2$. Consequently, we say that two elements belong to the same equivalence *class*, if they are semantically indistinguishable with respect to the properties used to define the equivalence relation. The earlier usage of class as a pattern describing the format, or decomposition, of an element is simply a special case of this more general interpretation of the class concept. The nature of a particular data denotational language governs what equivalence relations, and thus what classes, can be defined.

We assume that all data elements in DS belong to a class. We will also assume that the class of any element is invariant, although in the next section we will explore the possibility of permitting certain kinds of class migration.

If all data elements must belong to some class, then it is reasonable to expect that in set operations, such as $set1 \cup set2$, the elements must be, in some sense, conformable with respect to class. One such restriction might be that the elements of $set1$ and $set2$ belong to the same class. One should not be allowed to mix "apples" and "oranges". Well, perhaps this is too stringent. But, at least the resulting set should be one of class "fruit". In the following section, we discuss issues relating to the class of data elements in general, and the class of set elements in particular.

2.3. Class Inheritance

We have been using the Turing Machine model to develop the distinction between data as bit strings and their denotation by ordinal position or *uid*'s. However, it was assumed that the bit strings comprising elements of a persistent data space, DS, need not be interpreted in their entirety as either enumeration *uid*s or *integer*, *real*, or other environment dependent values. The class concept provides a formal mechanism for denoting, and accessing, substrings within *uid.bits*. We permit composite elements, such as records or tuples, whose component parts are denoted by value and element designators of the form *uid.attribute*. The collection of attributes that are defined on an element define the class of the element, in precisely the same way that the schema of a relation defines the class of its tuples.

In Section 2.1, we said that a *class* defines the well-formed expressions denoting data in the data space, DS. This we subsequently generalized in terms of an equivalence relation. Let us develop a concrete example in which the property defining an equivalence *class* is simply assumed to be a set of attributes, perhaps established by a statement such as

```
PERSON  isa  CLASS
       having  { name, age, social_security_nbr }.
```

All elements belonging to the class PERSON are now known to have the attributes *name*, *age*, and *social_security_nbr* defined on them.[25] So, if *x* denotes such an element, the value designator *x.name* is well-formed. We might want some elements to have other attributes, as well. A doctor might have the additional attributes *specialty*, *training*, and *address*; or a patient might have the additional attributes *case_history*, *address*, and *outstanding_amount_due*. The class of elements representing these kinds of people might be established as *subclasses* of the class PERSON by the statements

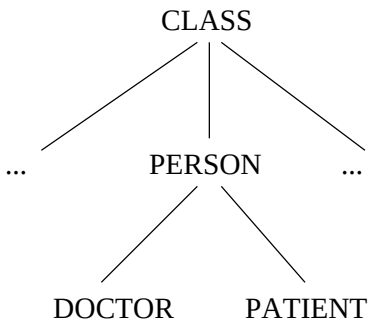
[25] To the data space these attributes are just bit strings; it is unconcerned with whether they are integer, real, character or whatever.

```
DOCTOR   isa  PERSON
       having  { specialty, training, address }

PATIENT  isa  PERSON
       having  { case_history, address, outstanding_amount_due }
```

We have refined the original class partition.

Every element belonging to the class DOCTOR, because it "isa" PERSON, has the attributes *name*, ..., *social_security_nbr*, as well, and so "inherits" these attributes. So, if *y* denotes an element of class DOCTOR, then *y.name* is well-defined data denotational expression as is *y.speciality*. Every attribute, or property, of a class must be inherited by any subclass of it; this is what we mean by *subclass*.[26] The resulting subclass hierarchy is often visually represented as a semi-lattice such as



A small subclass hierarchy
Figure 2-2

We would note (1) that any data element belonging to the class of DOCTOR also belongs to the class PERSON; (2) so that any expression for which an element of the class PERSON can serve as operand, an element of class DOCTOR can be operand as well; and (3) that restriction is created by defining *more* attributes on the elements of the subclass.

It is more difficult to define inheritance concepts on bit strings whose interpretation is environment dependent. In mathematics, the integers are a subset of the reals; an integer is a real

[26] Brachman [Bra83] correctly notes that inheritance as defined by the IS_A construct is really little more than a convenient syntactic shorthand for incrementally creating subclasses. We, too, will treat it in just this fashion. This way of incrementally creating subclasses has also been called "specialization" [HuK87].

number, and can therefore serve as an operand wherever a real is expected. This inheritance property is not reflected in the bit string representations of these types. One may not use an integer bit string where a real bit string is expected. The integer must first be explicitly coerced into a real form. This example becomes more pointed if we consider real numbers to be a subset of the complex numbers. The representation of data of type *complex* has a structure very different from that of type *real*. Both *integer* and *real* types can be coerced into a *complex* type, and so can strings of type *character*, such as "17.5" or "-3 + 5i". But, the very fact that coercion is necessary emphasizes that the types *as they are represented* do not have an inheritance structure.

Assertion 2.3: The presence of coercion in the semantics of a language demonstrates that the elements being coerced do not conform to a desired inheritance structure.

This is a primary reason for wanting to treat class and type as distinct semantic concepts, as we did in Section 2.1. Because classes embody denotational information — the nature of well-formed data expressions in DS — we may assume a language design that assures class inheritance. Because types embody information about the processing environment and its interpretations, expected inheritance properties may, or may not, be present.

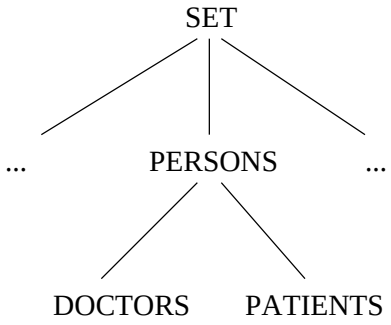
The issue of class inheritance becomes more interesting when we address sets. As observed in Section 2.2, a set in our model is an element of the data space, or Turing machine tape.[27] It has its own *uid*. It belongs to the generic class, SET. A more appropriate subclass of SET can be defined by class of elements that can comprise the set. For example, the classes, PERSONS, DOCTORS, and PATIENTS, might be declared by

```
PERSONS  isa  SET  of  PERSON  elements
DOCTORS  isa  SET  of  DOCTOR  elements
PATIENTS isa  SET  of  PATIENT elements
```

As declared above, there is no explicit inheritance between these three classes. Nevertheless,

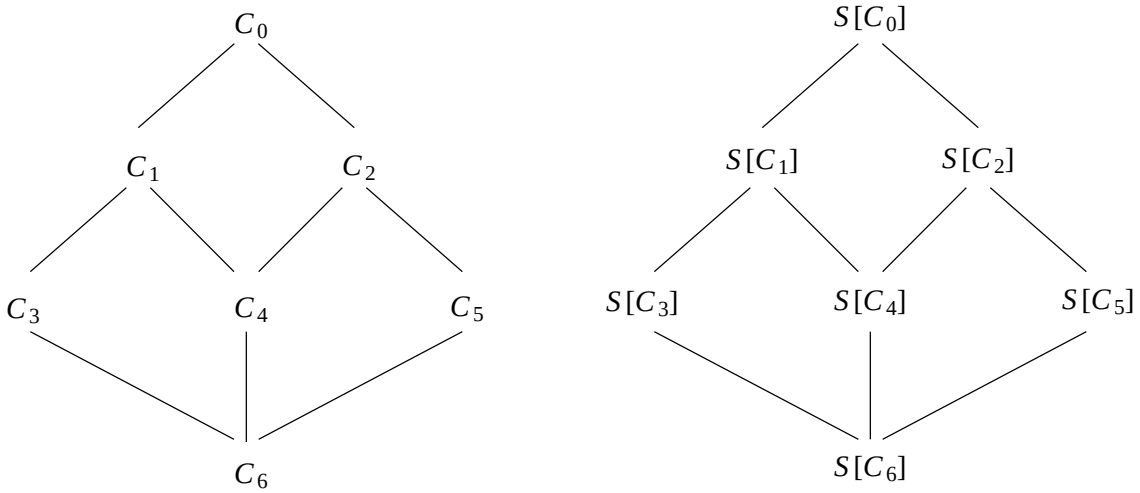
[27] We should emphasize that in our semantics, a *class* is not a *set*. We regard a class to be a only a pattern defining the structure of instantiated elements that could belong to the class. In object-oriented systems based on the smalltalk paradigm [GoR83], e.g. [BOS91, CoM84] the class denotes, and also manages, all its instantiated elements. While this is semantically possible, its practical implications in a distributed persistent implementation are daunting.

because an element in a set of class DOCTORS belongs to the class DOCTOR, and because an DOCTOR element can be linguistically employed wherever a PERSON element can be employed, we would expect that a set of DOCTORS can be used linguistically wherever a set of PERSONS can be used. That is, we have an implicit class hierarchy as shown below.



Induced Inheritance Semi-lattice of Set Classes
Figure 2-3

Using the more general notation of C_i for an arbitrary class and $S[C_i]$ for the class of sets consisting of elements in C_i one gets the apparent isomorphism shown in Figure 2-4. However, this illustrated isomorphism is deceptive; it is not necessary. It is possible to create consistent semantic models in which the inheritance lattice (or semi-lattice) of constructed classes is more



Class Inheritance Lattice and corresponding
Inheritance Lattice of Set classes
Figure 2-4

complex than that of their constituent elements. However, it can be demonstrated, under much more general definitions of the class concept than we have introduced here [PfF92], that if $S[C_i]$ is a sub-class of $S[C_k]$ then C_i must be a sub-class of C_k, or equivalently

Assertion 2.4: If sets of elements belonging to a specific class can be constructed, then a sub-lattice (sub-semi-lattice) of the set hierarchy must be isomorphic to the class hierarchy.

Of course, one may have sets of sets, and so forth. Adherence to the set class semantics introduced here eliminates many of the paradoxes arising in a pure naive set theoretic model.

The semantics of many standard set operators is relatively straightforward. For example, if *residents* denotes a set in the data space of class DOCTORS, then within the body of a looping construct such as

```
for_each  y  in  residents  do
       .
       .
end_loop
```

it is known that *y* denotes an element of class DOCTOR; and that any linguistic expression that is valid for this class, such as *y.name* or *y.specialty* will be semantically well-defined within the loop. Expressions such as *y.case_history* will not.

The semantics of set assignment also conforms to expected class inheritance hierarchy, but with an interesting twist. Suppose that *peer_group* denotes a set of class PERSONS. An assignment of the form

```
peer_group  <-  residents
```

is semantically well-defined because every element of the set *residents* must belong to the class DOCTOR, and hence to the class PERSON — its superclass. Consequently, any expression involving an element of *peer_group* will be well-defined. However, the assignment

```
residents  <-  peer_group
```

is not semantically well-defined because linguistic expressions, such as *y.speciality*, that are valid

for the class DOCTOR may not be defined for an arbitrary element in *peer_group* of class PERSON.

Consider the familiar numeric hierarchy involving integers and reals, and let i and x denote integer and real variables respectively. The assignments

```
x  :=  i
```

and

```
i  :=  x
```

are both syntactically legal in most programming languages, but only the former has semantics that really conforms to the type hierarchy. In this case, the actual value of i is assigned to x, albeit possibly with a different representation. In the latter, the value of x, unless by chance it is integral, will be altered and a *different semantic value* assigned to i. The value of x is coerced into an integer form by some environment dependent rule, such as truncation or rounding. These two examples lead to the observation that

Assertion 2.5: If the semantics of a language allows class (or type) inheritance, then inheritance must be monotone; only elements of a subclass may be used in lieu of elements of a superclass, without some form of *ad hoc* **coercion.**

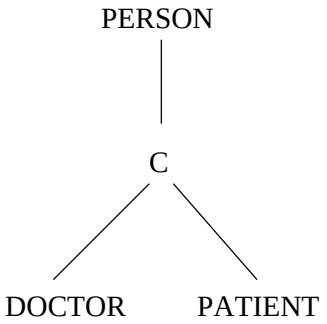
The semantics associated with the familiar union, $\cup$, and intersection, $\cap$, operators in a class hierarchy model are far more interesting. We will only sketch a few of the representative complexities in this report. If the two operands belong to the same class, or one is a subclass of the other, there are no semantic problems. But suppose the two operand classes are not directly comparable in the class inheritance lattice. Suppose, for example, that *residents* and *trauma_patients* are instances of sets of DOCTORS and PATIENTS respectively. We wish to look at each element in the union of these two sets, say by the looping expression

```
for_each  z  in  residents union trauma_patients  do
       .
       .
end_loop
```

The question is: what is the class of the expression *residents* $\cup$ *trauma_patients*? or more

particularly, what is the class of *z* within the loop body? A reasonable semantic interpretation is that, since the class of *residents* and the class of *trauma_patients* are both sub-classes of PERSONS, their union is of class PERSONS, and in particular *z* must be of class PERSON. These semantics, in which the class of a union operator is the least upper bound of its operand classes, can be implemented in fairly straightforward manner. But note, the expression *z.address* is well-defined regardless of whether *z* denotes an element in *residents* or in *trauma_patients*, because *address* was declared to be an attribute of both the classes DOCTOR and PATIENT. Yet *z.address* is not well-defined if one regards *z* to be of class PERSON.

It is possible to define a semantics in which the class of the union is the *most general* upper bound of its two operand classes. In this case, the class of *z* would be a new class *C* with attributes *name*, *age*, *social_security_nbr*, and *address*; the subclass hierarchy of Figure 2-2 would become



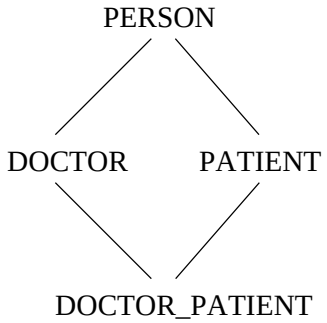
Generalized subclass hierarchy
Figure 2-5

and class of the union would be $S[C]$. The creation of such most general upper bounds in the subclass hierarchy has been called "generalization" [HuK87]. While it leads to a consistent semantic interpretation, it appears to be fairly difficult to implement in practice.

The intersection operation presents different problems. Suppose we wish to process the set of *residents* who have themselves been *trauma_patients*, say in a loop of the form

```
for_each  z  in  residents intersect trauma_patients  do
       .
       .
end_loop
```

What is the class of the set *residents* $\cap$ *trauma_patients*? What is the class of *z*? Readily, *z* must have all the attributes of DOCTOR as well as all the attributes of PATIENT, and thus be a subclass of both. It will be the greatest lower bound of the two argument classes. This implies the class hierarchy illustrated in Figure 2-6.



Closed subclass hierarchy
Figure 2-6

We call this a *closed* subclass hierarchy because it is closed with respect to set intersection operation.[28]

Establishing the class of the set resulting from an intersection operation is only part of the issue. More important are the semantics that determine the actual resultant set. Let *x* denote any element of *residents* belonging to the class DOCTOR and let *y* denote any element of *trauma_patients* belonging to the class PATIENT. Because the elements they denote belong to different classes, *x* and *y* cannot possibly denote the same element. Consequently, to be a member of the intersection, *z* must already be an instantiated[29] element of the class

[28] Even if the class DOCTOR_PATIENT has never been explicitly declared by the programmer, it could be artificially created. For example, in our case of class declaration which includes only associated attributes one can form the union of the set of DOCTOR attributes with the set of PATIENT attributes to generate the new class. But, this will prove to be of little value.

[29] Instantiation, in the object-oriented sense, is synonymous to enumeration in our model; a new element with a new *uid* is written to the data space, DS.

DOCTOR_PATIENT! For *residents* $\cap$ *trauma_patients* to be non-empty, the set *residents* must consist of some elements in the class DOCTOR_PATIENT, as must the set *trauma_patients*. The intersection operation then only extracts those DOCTOR_PATIENT elements which belong to both.

Assertion 2.6: In a persistent data space, where all elements have *uid*'s and all *uid*'s are bound to a single class on instantiation, the intersection operator is only a mechanism for designating a subset of existing elements.

Not only are these intersection semantics counter-intuitive, they also stand in clear contrast to the relational model in which the intersection (which can be generalized to create the important join operator) operator generates a new set of tuples using a value based concept of identity.[30] It is worth exploring these two different intersection semantics further.

Intuitively, there may be elements of type DOCTOR in a data space corresponding to doctors in the real world. Some of these may be residents and therefore in the set *residents*. Similarly, we might expect the periodic addition of PATIENT elements, and if they have suffered a trauma, insertion into the set *trauma_patients*. Subsequently, a resident, x, may suffer a trauma with a corresponding PATIENT element, y instantiated and entered into *trauma_patients*. A query to discover which residents have been trauma patients seems to be quite reasonable. Value based identity, that is $x.name = y.name$ and $x.social\_security\_nbr = y.social\_security\_nbr$, may be used (as in the relational join operator) to establish that the persistent data elements x and y actually denote the same entity *in the real world being modelled*, and thus should be combined. Could such an operation be accommodated in the model of persistent data that we have been developing? The answer seems to be a qualified, yes.

The governing condition of assertion 2.6 is that an element instantiated with respect to a class C, cannot later be bound to a class C'; e.g. an integer variable cannot subsequently be

[30] This creates additional problems with regard to any system-wide uniqueness of keys, since clearly tuples in the relations resulting from any of the operations must have identical counterparts with identical keys in the argument relations. However, resolution of this is not issue in this report.

declared to be real. If, instead, one permits individual elements in the data space to change their class, say by migrating to a subclass, then the assertion need not be true. One can easily visualize a join-like, intersection operation on elements of *residents* and *trauma_patients* as describe above in which two elements are considered to be equivalent on some appropriate basis, and in which one of the two argument elements, say x is moved into the intersection class and the attributes of the other assigned to it. But, now the semantics become messy. Should the element y be retained, or deleted. If it is retained, then the established identity is not retained in the persistent data space, we still have two distinct data elements. If it is deleted then all occurrences of its *uid*, say in sets or other denotational expressions, must be replaced by the *uid* of x.

As observed above, the relational model with its set semantics based on the tuple element bit string, not its ordinal identity, has no such problems with either intersection, or its extension as a natural join. The tuples denoting an individual as doctor, denoting an individual as patient, and denoting him as both, are distinct tuples. Doesn't a set semantics based on the element bit string, or content, make more sense? Possibly. However, implementing the union, $\cup$, operator which is known to be required in a complete query system, c.f. (p. 242) [Mai83], turns out to be quite difficult in a distributed environment, as shown by the following example. We may assume that there exist at least two relations r_1 and r_2 with the same schema, R. For simplicity, we assume that $R = \{a_1, a_2\}$, where a_1 is the key. If the property of being a key is restricted to individual relations, then both r_1 and r_2 below are well-formed relations

r_1 =

a_1	a_2
0	w
1	x

r_2 =

a_1	a_2
1	y
2	z

but $r_3 = r_1 \cup r_2$ must be the relation

```
          a1 | a2
r3 =    ----+----
          0 | w
          1 | x
          1 | y
          2 | z
```

in which a_1 has lost its key property. To be a *correct* implementation, the non-duplication of key values must be maintained for all relation instances in the database. Because of the cost, few existing relational database implementations will automatically eliminate duplicate keys in even a single relation. In a distributed system, the assurance of key uniqueness is even more costly.

Assertion 2.7: In distributed persistent data spaces, content based data element identity is computationally prohibitive.

This is the primary reason that we have elected to ignore content based element identity as a basis for set semantics.

This has been a long section. However, we believe its contents are crucial. In Section 1.2, because of the impossibility of uniquely naming every element in very large persistent data spaces, we asserted (1.5) that "any programming language which references persistent data must provide general mechanisms for creating, naming, and operating on sets of data" — and, in practice, all management of persistent data has been set based. Thus the semantics of such operations must be developed. As asserted in (2.2), "the fundamental semantic characteristic of any programming language that manipulates sets is whether they are considered to be sets of distinct identifiers denoting data elements, or sets of distinct element bit strings". This choice governs the semantic nature of any language for processing persistent data. If, because content based element identity is impractical in distributed persistent data spaces (2.7 above), one chooses the former and if one also categorizes such elements with respect to a class hierarchy, then there will be semantic restrictions on what such a programming language can, or cannot, do. Assertion 2.6 illustrates one such restriction.

3. Persistent Name Spaces

Central to any programming language is the way that a user may name those items of interest to him, typically the nouns of the language. In this report we are concerned with ways of denoting elements of the persistent data space, DS, with symbolic strings, or names. Naming is not the only mechanism for identifying persistent data elements [KhC86], but it is a primary one.

As in any programming language, a symbolic data name must be bound to the element it denotes, or equivalently in our model to its *uid*. But, when persistent data is named, both the name itself and the binding must also be persistent, as asserted in (1.5). Persistent names cannot be freely reused in different source program files, or even within different lexical blocks of the same program file, as most programmers are used to doing. One must treat persistent names as global names, that are global to not only a single process, but global to all processes which operate over the shared persistent data space.[31] The consequence is that our space of symbolic names seems to be far too small to facilitate naming all of the elements of a very large data space with many disparate kinds of data. Accumulating large collections of similar data items into sets helps, but we still seem to exhaust the range of appropriate names. The purpose of this section is to briefly consider four different ways of expanding a programmer's name space. They are:

(1) allow arbitrarily long symbolic names;

(2) partition the name space so that the same name in different partitions may have different denotational bindings;

(3) subscript names; and

(4) parameterize names so that the same name with different parameters may be bound to different elements.

---

[31] Linda [CaG89, Gel85] is a programming language that operates over a persistent data space according to our definition. It is one that is concerned with global naming [FlH89]. However, it does not have a name space that is conceptually separate from its data space; individual tuples of the tuple space cannot be explicitly named. Instead, it employs content based identity to denote desired tuples.

The first two techniques are embodied in an operating system's directory structure. Persistent files are denoted by path names of arbitrary length. Of course, very long symbolic names are an invitation to error. The *cd*, or change directory, command can be regarded as nothing more than a mechanism for designating a common prefix for very long path names in an otherwise conceptually flat name space. Now only a relatively short symbolic suffix need be provided to derive the entire path name.

Directory systems also serve to partition the name space. The same "name" or symbolic suffix will denote different files if their complete path names have different prefixes. Moreover, certain prefix strings may be reserved for specific users or for specific groups of users through familiar protection mechanisms. Name servers in distributed systems employ both these techniques [Lam86], but so may a language designed to operate over a persistent data space. If the language provides for a partitioned persistent name space, then one need not be constrained to a strictly tree-structured configuration [PFW88], and, perhaps more importantly, one can write processes within the language itself to search the common name space. In the database literature this is sometimes called "browsing".

Mathematicians routinely use subscripting to distinguish between similar, but distinct, elements. So too, do all familiar programming languages. But, subscripting in traditional programming languages has come to be limited to array structures. One can conceive of languages in which all user generated names, such as those of processes, can be arbitrarily subscripted in order to distinguish between different instantiations. Such general subscripting facilities can be effectively implemented [PfF90], but the development of any particular syntax or implementational semantics is not the point of this report.

The concept of parameterizing individual names so that their meaning, or binding, is determined by the particular parameters provided at the time of reference has not to the author's knowledge been actively studied with regard to higher level languages. Assembly level macro

substitution represents a limited form of such a facility.

The four techniques described above probably do not exhaust the available mechanisms for organizing and expanding a persistent name space, but they do support the contention that

Assertion 3.1: To program over persistent data spaces, more sophisticated methods for creating and manipulating persistent names must be employed than are found in currently implemented programming languages.

4. Data Deletion

From time to time a user may decide that persistent data is no longer necessary and can be deleted; but deletion of data, even of non-persistent data[32] is complex and difficult. In our Turing machine model, data elements are never deleted from the DS tape because *uid*'s denote an enumerated position and cannot be reused. Since the DS tape is unbounded, deletion is, in theory, not needed. But in practice, storage is bounded. Need we delete items? Must we make provision to reuse *uid*'s?

The latter is easiest to answer. In a practical implementation we will allocate only a fixed number of bits to represent *uid*'s. If we treat *uid*'s as 64-bit integers, we have 1.84×10^{19} distinct *uid*'s which according to some estimates approximates the number of atoms in the universe. If this still seems insufficient, one could employ 128-bit *uid*'s to obtain 3.4×10^{38} unique elements, while no more doubling the physical storage requirements. Although technically finite, there should be no need to ever reuse a *uid* once its data element has been enumerated.

Virtually all implementations of persistent data will delete data items in order to reuse physical storage. But, in a distributed system to which very high capacity storage devices can be added continually, we might conceive of an implementation in which physical storage is never exhausted and in which data elements are never deleted. Even if physical storage were not a

[32] Either the extensive literature on garbage collection in LISP systems, or the detailed discussion of the `delete` operators and destructors in C++ where one encounters expressions such as "...the deletion of an object may change its value" [Str87] (p. 500) can be regarded as cases in point.

constraint, what seems to inexorably force consideration of data deletion is the psychological finiteness of a user's name space. In practice, we delete an item because it is no longer of value, and because we want to *reuse its name* for new data.

We will say an item is *deleted* if its *uid* no longer denotes a valid string. In our Turing machine model, an unwanted item will be deleted by rewriting the element with a special *deleted* symbol.

Under what conditions can a data element be deleted? Certainly, a data element *uid* that is a member of an undeleted set in DS should not be deleted. Indeed, no element to which there is a valid reference in either the data space DS, or name space NS, should be deleted. One mechanism for controlling unwanted deletion is by means of reference counters. These are an old device, which the author first encountered in 1963 [Wei63], although their use may predate even this. Essentially a reference counter is an integer attribute, $uid.ref\_cntr$, associated with every *uid*.[33] It is initially zero, and incremented every time a new reference to *uid* is entered into DS, e.g. by a set operation, or into NS, e.g. by binding a name to the *uid*; and decremented every time a reference is erased. Conceptually, the data element denoted by *uid* may be deleted whenever $uid.ref\_cntr = 0$. However, in practice, more care must be taken. Because a process may temporarily cache a *uid* in its local storage, it is possible to have $uid.ref\_cntr = 0$, even though there are still valid references to it.

We can construct a deletion semantics as follows. Let $ds\_delete(uid, P_j, t_k)$ be an operator which, if $uid.ref\_cntr > 0$ at t_k has no effect, but which, if $uid.ref\_cntr = 0$ at t_k has the effect of inhibiting any reference to *uid* by any process initiated after t_k so long as $uid.ref\_cntr$ remains equal to 0. Now, one can show that

Assertion 4.1: If $ds\_delete(uid_i, P_1, t_1)$ **has been issued, and if at** t_1**,** $uid.ref\_cntr = 0$**, and if at** $t = \max_{t_k}\{p\_terminate(P_j, t_k) \mid p\_initiate(P_j, t_{k'}), t_k < t_{k'}\}$ $uid.ref\_cntr = 0$**, then the data element** *uid* **can have no references within the data space, DS, the name**

[33] The *ref_cntr* attribute may be regarded as a property of both generic classes, CLASS and SET.

space, NS, nor any executing process and so can be actually deleted.

Note that the third condition for these deletion semantics is precisely the condition for persistence given in assertion 1.2.

An unfortunate corollary to the preceding assertion appears to be

Assertion 4.2: If *x* is a persistent element name appearing in a program, and if the preceding deletion semantics are to be supported, then *x* can not be bound to a *uid* at compilation, but must be rebound (if possible) on every execution through lookup in the name space, NS.

Readily, this assertion implies a painful performance hit and one might expect many compiler implementations to ignore it, trusting that good programming, or good fortune, will prevent reference to a deleted *uid*.

We do not know if the deletion semantics described here are necessary in programming languages that manipulate persistent data. They may not be. But this discussion does emphasize the kinds of differences that exist between programming languages whose data is only assumed to exist while the process itself is in scope, and those which may directly reference persistent data.

5. Summary

In the preceding, we have employed a Turing model on which to base the semantics of persistent data — elements on a shared persistent data tape, which once enumerated can not be erased and which can only be changed by the deliberate rewriting by a process (1.1). It leads naturally into the distinction between the elements of the tape, or data space, and their ordinal position, or *uid* (1.3).

Because an effective programming language must employ symbolic names that are bound to *uid*'s, and because such a name space must also be persistent (1.4), we came to the unsurprising conclusion that such languages must be vitally concerned with the denotation and manipulation of sets of elements (1.5). Much of Section 2 then revolved around the semantics of such sets

of persistent elements. These semantics must be governed by whether one chooses to regard a set as a set of element identifiers, *uid*'s, or a set of element bit strings themselves (2.2). The relational model of data employs the latter. Because of complexity of implementing the $\cup$ operator (2.7), we chose to develop our semantics using *uid*'s. In this regard, it is similar to several object-oriented models.

Knowing that developing set operation semantics based on a naive untyped set model would be extremely difficult, we first introduced the concept of element classes and explored the distinction between data types which are dependent on the processing environment and element classes which properties of the persistent data space itself (2.1). Given a class concept, it is natural to introduce the powerful concept of class hierarchies and inheritance. However, we observed that, given the standard representations of numeric types, natural type inheritance must frequently be enforced by *ad hoc* coercion (2.3); it is not a property of the data itself.

Finally, if all data elements belong to classes in a class hierarchy, then sets of elements must also belong to a parallel class hierarchy which mirrors the structure of their constituent element classes (2.4). These class concepts combine to restrict the semantics of any resulting language. This was illustrated by the monotonicity of inheritance (2.4) and the necessary semantics of set intersection when the operand sets belong to non-comparable classes (2.6).

Section 3 discussed four ways by which the persistent name space of such a language could be expanded to facilitate symbolic reference by many user processes to the shared, arbitrarily large, persistent data space. And the thorny, incompletely resolved, issue of data deletion, which is forced on a language designer by the need to reuse symbolic element names, was examined in Section 4.

6. Syntax and Implementation

A consistent syntax for the semantic model proposed in the preceding sections can be created. All of the code fragments found in Section 2 were drawn from a working database system, called ADAMS [PSF88, PFG91], that is largely based on this model. The syntax of the ADAMS language provides for (1) a clear distinction between the data type and element class concepts, based on value designator and element designator constructs, (2) multiple inheritance in the class hierarchy, (3) general set expressions involving sets in the data space, (4) process control, including both data space and host process statements, based on iteration over sets in the data space, (5) general assignment operators between the various components of the persistent data space and the variables of the local processing environment, (6) a hierarchical name space which allows any element name to be subscripted and with which any process can dynamically interact, and (7) except for a very small fraction of the statements and expressions, static type checking in the preprocessor.

But, as Pratt points out [Pra84], there can be many equivalent programming syntaxes for the same semantic model; the virtual equivalence of the standard procedural programming languages demonstrate this. The existence of ADAMS only establishes that this semantic model can be made linguistically consistent. It is possible, and in fact probable, that clearer, better syntactic formulations can be devised.

A more important issue in the design of languages is whether the semantic model can be effectively implemented. ADAMS has been implemented over a loosely coupled network of heterogeneous SUN workstations and over a tightly coupled Intel iPSC/2 using the run-time support provided by the Mentat concurrent processing system [GrL90, Gri90]. The various applications that have been coded in this environment establish that the semantic model presented here can, in fact, be implemented to provide a shared, distributed, persistent data space with reasonable storage and processing overhead costs [PFG92].

We might, however, add in postscript that many individuals find programming over persistent data spaces quite foreign. There can be a considerable period of adjustment. It is the persistent binding of names and other program constructs that gives the most difficulty. For example, if we have the statements

```
adam  instantiates_a  PERSON
eve   instantiates_a  PERSON
```

in a program, then because the names *adam* and *eve* have been persistently bound to to the *uid*'s of newly enumerated data elements, these statements and hence the program itself can only be executed once! But we are used to writing code that doesn't quite work as we expected, rewriting it and re-executing it.

There must be a change in programming style. One change is to separate those programs which enumerate new named data elements, from those that only reference and manipulate existing elements. A second programming change is to avoid persistent naming as much as possible, and rely on navigation through the data space to access desired elements. For example, suppose that the following fragment of code has been executed

```
first_family  instantiates_a  PERSONS
parents       instantiates_a  MAP
              with image  PERSONS
```

Both $first\_family$ and *parents* are persistent names denoting a set of PERSON elements and a functional map whose image is the *uid* of some (unspecified) set of PERSON elements.[34] One can now execute the following fragment of code in which x, y, z and $folks$ are *variable* element names and so do not result a persistent name-uid binding.

[34] In ADAMS we make the distinction between *attribute* functions whose image is a data element that must be interpreted with respect to a particular environment, e.g. integer, real, string, and which we call a *value*, and *map* functions whose image is a *uid* denoting an element belonging to a known class. C.f. assertion 1.3.

```
ADAMS_var  x, y, z, folks

x  instantiates_a  PERSON
x.name  <-  'adam'
insert  x  into  first_family
y  instantiates_a  PERSON
y.name  <-  'eve'
insert  y  into  first_family

z  instantiates_a  PERSON
z.name  <-  'abel'
folks  instantiates_a  PERSONS
insert  x  into folks
insert  y  into folks
z.parents  <-  folks
insert  z  into  first_family
```

One can now navigate in the persistent data space with code such as

```
ADAMS_var  x, y

for_each  x  in  first_family  do
       printf ("%s tparents { ", x.name);
       for_each  y  in  x.parents  do
              printf ("%s ", y.name);
              end_do
       printf ("} n")
       end_do
```

to yield output such as[35]

```
adam   parents { }
eve    parents { }
abel   parents { adam eve }
```

To many programmers, writing code such as this seems quite unnatural. That is precisely the point of this report. Programming over a persistent data space, whether one uses the syntax and constructs of ADAMS or not, is very different from programming over volatile data.

For another example of the difference between programs involving persistent data and those which declare volatile data, consider the declaration of the class of a persistent data element. Since a *class* is only a linguistic construct declaring which data denotational expressions are well-formed, it is really only necessary at compilation.[36] They can be incorporated as a prefix to

---

[35] Since *first_family* and *x.parents* are sets, there are no guarantees regarding the order in which elements of the set will be encountered. It need not be as shown!

[36] This statement assumes static type checking, if run-time type checking is necessary, class information must be available at run-time as well.

every compilation in the manner of .h files in a C program, the DATA section of a COBOL program, or the type declarations of a Pascal program. But, they too must be persistent.[37] Such an approach of prefixing class declarations is viable, even with persistent data; but in ADAMS, class declarations are incorporated into the persistent data space, just as persistent data name bindings are. They are not redeclared with every new compilation. This engenders a new approach to writing code, in which the programmer actively searches the persistent data space to discover which persistent data names in his name space have been used, and which are available; what classes exist, and what properties they have.

It turns out that programming over a persistent data space is very different from traditional programming involving transient data; both in theory and in practice.

7. References

[AbK89] S. Abiteboul and P. C. Kanellakis, Object Identity as a Query Language Primitive, *Proc. 1989 ACM SIGMOD Conf. 18*,2 (June 1989), 159-173.

[AgG89] R. Agrawal and N. H. Gehani, ODE (Object Database and Environment): The Language and the Data Model, *Proc. 1989 ACM SIGMOD Conf. 18*,2 (June 1989), 36-45.

[AhU79] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, Reading, MA, 1979.

[AtB87] M. P. Atkinson and O. P. Buneman, Types and Persistence in Database Programming Languages, *Computing Surveys 19*,2 (June 1987), 105-190.

[Bee90] C. Beeri, Formal Models for Object Oriented Databases, in *Deductive and Object-Oriented Databases*, W. Kim, J. M. Nicolas and S. Nishio (editors), Elsevier Science Publ., North-Holland, 1990.

[Bra83] R. J. Brachman, What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks, *COMPUTER 16*,10 (Oct. 1983), 30-36.

[BOS91] P. Butterworth, A. Otis and J. Stein, The Gemstone Object Database Management System, *Comm. of the ACM 34*,10 (Oct. 1991), 64-77.

[37] While element classes must be persistent, they need not be static. But providing dynamic class declarations is very implementation dependent, and beyond the scope of this report. For details, see [PFG92].

[CaW85] L. Cardelli and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys 17*,4 (1985), 471-522.

[CaG89] N. Carriero and D. Gelernter, Linda in Context, *Comm. of the ACM 32*,4 (Apr. 1989), 444-458.

[Cod70] E. F. Codd, A Relational Model for Large Shared Data Banks, *Comm. of the ACM 13*,6 (June 1970), 377-387.

[Com79] D. Comer, The Ubiquitous B-Tree, *Computing Surveys 11*,2 (June 1979), 121-137.

[CoM84] G. Copeland and D. Maier, Making Smalltalk a Database System, *Proc. SIGMOD Conf.*, Boston, June 1984, 316-325.

[FlH89] C. J. Fleckenstein and D. Hemmendinger, Using a Global Name Space for Parallel Execution of UNIX Tools, *Comm. of the ACM 32*,9 (Sep. 1989), 1085-1091.

[Gel85] D. Gelernter, Generative Communication in Linda, *Trans. Prog. Lang and Systems 7*,1 (Jan. 1985), 80-112.

[GoR83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, MA, 1983.

[GrL90] A. S. Grimshaw and E. Loyot, The Mentat Programming Language: Users Manual and Tutorial, Dpt. of Computer Science TR-90-08, Univ. of Virginia, Apr. 1990.

[Gri90] A. S. Grimshaw, The Mentat Run-Time System: Support for Medium Grain Parallel Computation, *Proc. 5th Distributed Memory Computing Conf.*, Charleston, SC, Apr. 1990.

[GrG83] R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1983.

[HeP90] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.

[Hoa78] C. A. R. Hoare, Communicating Sequential Processes, *Comm. of the ACM 21*,8 (Aug. 1978), 666-677.

[HoU79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[HuK87] R. Hull and R. King, Semantic Database Modeling: Survey, Applications, and Research Issues, *Computing Surveys 19*,3 (Sep. 1987), 201-260.

[HWW91] R. Hull, S. Widjojo, D. Wile and M. Yoshikawa, On Data Restructuring and Merging with Object Identity, *IEEE Trans. on Data Engineering 14*,2 (June 1991), 18-22.

[KhC86] S. N. Khoshafian and G. P. Copeland, Object Identity, *OOPSLA '86, Conf. Proc.*, Sep. 1986, 406-416.

[Kim90] W. Kim, Object-Oriented Approach to Managing Statistical and Scientific Databases, in *Statistical and Scientific Database Management*, Z. Michalewicz (editor), Springer-Verlag, Berlin-Heidelberg-New York, Apr. 1990, 1-13.

[KoS86] H. F. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, New York, 1986.

[Lam86] B. W. Lampson, Designing a Global Name Service, *Proc. on Distributed Computing*, 1986, 1-10.

[Mai83] D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, MD, 1983.

[PFW88] J. L. Pfaltz, J. C. French and J. L. Whitlatch, Scoping Persistent Name Spaces in ADAMS, IPC TR-88-003, Institute for Parallel Computation, Univ. of Virginia, June 1988.

[PSF88] J. L. Pfaltz, S. H. Son and J. C. French, The ADAMS Interface Language, *Proc. 3th Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan. 1988, 1382-1389.

[PfF90] J. L. Pfaltz and J. C. French, Implementing Subscripted Identifiers in Scientific Databases, in *Statistical and Scientific Database Management*, Z. Michalewicz (editor), Springer-Verlag, Berlin-Heidelberg-New York, Apr. 1990, 80-91.

[PFG91] J. L. Pfaltz, J. C. French and A. Grimshaw, An Introduction to the ADAMS Interface Language: Part I, IPC TR-91-06, Institute for Parallel Computation, Univ. of Virginia, Apr. 1991.

[PFG92] J. L. Pfaltz, J. C. French, A. S. Grimshaw and R. D. McElrath, Functional Data Representation in Scientific Information Systems, *Intern'l Space Year Conf. on Earth and Space Science Information Systems (ESSIS)*, Pasadena, CA, Feb. 1992.

[PfF92] J. L. Pfaltz and J. C. French, Multiple Inheritance and the Closure of Set Operators in Class Hierarchies, IPC TR-92-004, Institute for Parallel Computation, Univ. of Virginia, June 1992.

[Pra84] T. W. Pratt, *Programming Languages: Design and Implementation, 2nd Ed.*, Prentice Hall, Englewood Cliffs, NJ, 1984.

[RiC87] J. E. Richardson and M. J. Carey, Programming Constructs for Database System Implementation in EXODUS, *Proc. ACM SIGMOD Conf. 16*,3 (Dec. 1987), 208-219.

[RiC89] J. E. Richardson and M. J. Carey, Persistence in the E language: Issues and implementation, *Software—Practice & Experience 19*,12 (Dec. 1989), 1115-1150.

[SaG90] K. Salem and H. Garcia-Molina, System M: A Transaction Processing Testbed for Memory Resident Data, *IEEE Trans. on Knowledge and Data Engineering 2*,1 (Mar. 1990), 161-172.

[ScY89] P. Scheuermann and C. Yu, editors. Report of the Workshop on Heterogeneous Database Systems, NSF Report, Northwestern Univ., Evanston, IL, Dec. 1989.

[Sch77] J. W. Schmidt, Some High Level Language Constructs for Data of Type Relation, *Trans. Database Systems 2*,3 (Sep. 1977), 247-261.

[SDD86] J. Schwartz, R. B. K. Dewar, E. Dubinsky and E. Schonberg, *Programming with Sets: An Introduction to SETL*, Springer-Verlag, New York, 1986.

[SeL90] A. P. Seth and J. A. Larson, Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases, *Computing Surveys 22*,3 (Sep. 1990), 183-236.

[Son89] S. H. Son, Recovery in Main Memory Database Systems for Engineering Design Applications, *Information and Software Technology 31*(Mar. 1989), 85-90.

[Str87] B. Stroustrup, *The C++ Programming Language*, Addison Wesley, Reading, MA, 1987. (Second edition, 1991).

[TTC90] G. Thomas, G. R. Thompson, G. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox and B. Hartman, Heterogeneous Distributed Database Systems for Production Use, *Computing Surveys 22*,3 (Sep. 1990), 237-266.

[Wei63] J. Weizenbaum, Symmetric List Processor, *Comm. of the ACM 6*,8 (Sep. 1963), 524-536.

Table of Contents