The Probe Project

Tim Blanchard†

John A. McDermid

Department of Computer Science,
University of York

tim@uk.ac.york.minster

January 1992

† Research is sponsored by SERC/UK research grant number 89556293 and with the support of the Procurement Executive Ministry of Defence.

Abstract

This report presents a detailed overview of the issues concerning the construction of an object database system called Probe. Probe is based on the Ten15 persistent programming language, exploiting its strong type system and flexible persistence mechanism to build an efficient and malleable database system.

This document describes the Probe system and identifies the criteria for a successful database programming language. The success to which Probe matches these ideals is discussed.

Probe is a based on a hierarchy of Ten15 data structures: data structures which manipulate the persistent store; which are used as a basis for database and bulk data structures; and a programming notation which provides a seamless programming environment for the abstract machine that Probe provides. A section of this document makes particular reference to the conceptual programming language for the system, called TDBPL.

1. Introduction

Databases and programming languages have remained diverse fields of research up until the development of persistent programming languages [8] in the early 1980s. Persistent programming languages provide a perfect vehicle with which to research the field of database programming. In this report, we describe an attempt to merge databases and persistent programming languages culminating in the Probe database programming language.

In this section is to illustrate the problems of existing approaches to database programming in order to provide the context and motivation for our work.

There have been two major fields of research into the construction of database programming languages (**DBPL**). One approach has been to integrate a particular data model into the language. The second approach has been to exploit the characteristics of persistent programming languages (**PPL**).

1.1. Model-based approach

The first approach in developing DBPLs has been to add a specific database model to a programming language by providing the data structures associated with the model as primitives within the language. Each database model has been integrated into a number of programming languages. For example, the relational model [17] has been embedded into Pascal [34] and Modula, and a language called DBPL has been developed specifically to support the relational model [30]. The functional model has been embedded into ADA, producing the language ADAPLEX [35], and the persistent programming language, Galileo [3]. The object oriented model has been widely researched, with such languages as Alltalk [36], O++ [2] and E [33] being developed.

The model-based approach draws its popularity from the fact that a clearly defined model is present within the language and the functionality of the original language still exists. The main problem of this approach is that the actual implementation of the model and the representations of the data on disk are hidden from the programmer by *access expression queries*. This means that the programmer has no flexibility to optimize or improve the functionality of the model. This is the main reason that we have not followed this route in our research.

1.2. Persistent programming language approach

The second approach to DBPL development stems from the development of persistent programming languages in the early 1980s. Persistence is the ability for arbitrarily complex program data structures to outlive the execution of a single program. Persistence relies upon a storage medium, such as a magnetic disk, being addressable and structurable from primitives provided in the programming language which implements it. In addition, the type and structure of the data are also stored. Languages have been developed that directly support persistence, such as PS-algol [7], Napier88 [31] and Ten15 [11].

The main feature of PPLs is that the backing store is represented as an abstract entity within the programming language. This means that arbitrarily complex data structures can be built up and preserved on the persistent store without loss of type or structure. What this means for databases is that the data structures of a particular model can be built up, preserved and manipulated by the operators that support persistence.

There are several advantages to using the persistent programming language approach to database development. Firstly, the implementation and persistent representation of the database is visible. This means that the database can be altered to tailor performance or to extend its application domain and so it can be used as a basis for the specialisation of the database model. Secondly, by using persistent data structures, the persistent database is not constrained to one data model. This means that data structures can easily be created, based on existing data structures, which introduce other models into the language. A third advantage is that it is possible to ignore the database model for data storage since a mechanism already exists in the language to allow for the persistence of data. This might be necessary in cases where the data that needs to be stored cannot be expressed in a database model, or where the extra cost of allowing for the generality of the modules which implement the database detracts from the performance of the system.

We feel that this approach can lead to inefficiencies within such a system, since the low level optimizations are not available to the programmer since the disk and memory cannot be accessed directly. However, developing using a high level language speeds up the development process, since abstract ideas can be expressed more readily, the type system prevents any run-time type errors and it facilitates the adaption and maintenance of the resulting software. There is a trade off between the ease of system development and efficiency of the resulting system. We feel that the ease of development is of utmost importance. Efficiency of the system can be achieved by introducing optimizations into the model of the system, and these are discussed later in this paper.

However, there are further problems associated with this approach to database programming. These can be generalised into two main areas: those concerned with the management of bulk data, and the effects of concurrency.

1.2.1. The management of bulk data

Bulk data is the aggregation of data of a similar type or value or relationship. Such aggregators include sets, tuples and lists. Two main problems occur with regards to the management of bulk data: how to represent it, and how to make access to it efficient.

First, there is typically an impedance mis-match [19] between the intended database model and the type system of the host programming language. Depending upon the particular model used, it may require relational tables for the relational model, sets, tuple and objects for the object oriented model or functional mappings for the functional model. However, programming languages typically have structures and arrays as their main data constructors, using pointers to build up more complex data structures. Hence the meaning of the programming language data structures have to be overloaded to represent the data structures required by the database model. Provided that a mechanism is present that allows the representation of the data structure to be hidden, such as abstract data types, existential values or lambda type functions, then PPLs can simulate database models, although the simulation may be complex and clumsy.

The main problem with the simulation technique is that not only must the semantics of the database type constructors be captured but, in addition, an efficient representation must be maintained on the persistent store. This leads onto the second problem, concerning the management of bulk data. Not only must the PPL support the efficient management of data on persistent store, but it must also provide a mechanism which has uniform performance for all extremes of stored data, from very small to very large. To this end, a representation based on the decomposition of the data structure into a tree suggests itself as a mechanism flexible enough to support all variations in sizes of data, but, as a result, there is an increase in application paradigm development complexity, and the source code becomes progressively more complex to understand and maintain. This goes against the original rationale for persistent programming languages: to minimise the time it takes to write the source code which deals with the interaction between the language and the backing store [7].

Another problem with the management of bulk data is that a generic storage structure mechanism takes no account of the nature of the data that it stores, hence the characteristics of data, such as constancy or sparsity (e.g. in a sparse matrix) are ignored. Optimized storage capabilities can be provided as an alternative to the generic mechanism depending on analysis of the data as it is incrementally added to the database constructor.

1.2.2. Concurrency

There are two problems associated with concurrency in DBPLs: the general problems of concurrency control and the linguistic differences between PPLs and DBPLs.

The first problem for using PPLs as database programming languages concerns the concurrent nature of databases. Databases that are restricted to a single user are of limited use. By permitting several processes to have concurrent access to a central database, the usefulness of the database increases dramatically.

When adding concurrency and distributed databases to PPLs, several problems are encountered. The main problem is that persistent data is updated on an object-by-object basis, which conflicts with the database notion of a number of values updated at one instant in the form of a transaction. This creates problems with regards to implementing the object database model since, due to its hierarchical nature, values from the leaves to the root must be committed in turn, introducing the possibility of inconsistencies should a failure occur before every entity in the database has been updated. Additional problems include the lack of timing constraints over a process' access to persistent data and possible deadlock over shared persistent resources. This is the problem classically addressed by transactions in relational DBMSs.

The second problem concerns the linguistic basis for PPLs. The majority of PPLs are procedural-based: statements are executed in turn, and persistent data is committed to the persistent store sequentially. The problem with procedural programming is that no allowance is made for concurrent access to data on the persistent store, nor for failure and recoverability. In the first case, a procedure may read data from the persistent store, act upon it, then wish to rewrite the updated value. If locking of objects is not present, then the procedure which rewrites its value may overwrite a value recently updated by another process, or invalidate the data read in by another procedure which has an intent to write. In either case, the procedure-based approach is unable to cope with recovering the consistency of the persistent object. If object locking and process blocking are added, then the validity of a persistent object can be guaranteed. However, for large procedures, which update a large number of values in one go, it may take a long time to complete the transaction since it is waiting upon a large number of values to become free, and the chance of deadlock are increased.

A solution to these problems is to change the programming style of PPLs from procedure-based to transaction based, where procedures are broken down into a number of small, logically complete transactions, in order to minimize the contention for values and the amount of blocking required. Again, this will add complexity to the program development, and force programmers to adopt a more restricted view of the persistent store, indeed forcing them to be aware of the nature of the persistent store and of concurrency within the system.

1.3. Objectives

Persistent programming languages, despite the faults highlighted above, make an effective development platform for databases because of the availability of the backing store from within the language. The rest of this paper concerns the development of a database kernel in the persistent programming language Ten15, and discusses how the kernel can be used to implement an object database model. Specifically, our approach addresses problems related to the effective representation of database data structures on PPLs, and how these data structures can be managed efficiently.

The next section concerns our requirements for an object database model. The requirements are analysed, and it is suggested, in section three, that Ten15 provides the type system and persistence model appropriate to allow efficient implementation. Section four introduces Probe, the object database built on Ten15, describes the rationale for its design and analyses the component of its architecture. Section five presents our conclusions and ideas for future work.

2. Requirements for an object database model

The type of database systems which we are interested in has a number of basic criteria. First, it must have a general application domain, with the ability to support applications such as IPSE, CAD, multimedia databases, as well as the more traditional database applications, such as personal data storage and stock control. Second, the system must have efficient support for the management of bulk data.

To this end, the *object database model* was chosen as the data model to be implemented on the system. It is eminently more flexible than more traditional data models, such as the relational model or functional model, for such a diverse range of application domains. The object database model is a more general form of the object oriented database model. Where it differs is that an object is merely a collection of data, which may be first class procedures which can act as methods. Access and update of data within an object is achieved by an external navigational mechanism. This model is similar to the structural object oriented database model, as exemplified by Damokles [22].

To support the object database model efficiently, we have the following requirements for an object management system:

- A rich data model, which allows generalisation and aggregation of objects [26], arbitrary nesting of objects and direct support for object identity or object sharing through a simple navigational scheme.
- Support for persistence, where any object has the right to be persistent, regardless of its type or structure.
- Concurrent access to objects, allowing for the security and integrity of objects to be protected when concurrent accesses are made to shared objects.
- Support for bulk data by providing means of decomposing large objects into smaller entities when they are preserved on disk, and using mechanisms such as clustering and indexing to enable portions of the large object to be accessed and updated efficiently.

  In addition, support should be scalable. There should be no penalty in performance for differing extremes in size of bulk data, in that access to, say, a small set, is not hindered by optimization mechanisms that must be included to make access to large sets efficient.

- Provision of a reasonable mechanism for the incorporation of new data types, storage mechanisms and algorithms, and the optimization of existing data structures to cater for changes in usage.

Our five requirements listed above can be used to investigate the suitability of persistent programming languages in the implementation of database models. Section 3 analyses the requirements, then suggests that Ten15 is a suitable PPL for the development of such an object manager.

3. Ten15

Of the five features mentioned in the previous section, three – a rich data model, extensibility and support for bulk data – are reliant on the expressiveness of the type system of the PPL. Concurrency and persistence are features of the language which are drawn from a particular programming paradigm. Hence our ideal model can be broken down into three main areas: *persistence*, *type system* and *concurrency*. Each of these areas is analysed in turn with respect to the language Ten15 [11, 24].

Ten15 is the culmination of many years' research at the Defence Research Agency (formerly the Royal Signals and Radar Establishment), Malvern, UK into the development of a strongly typed intermediate language. The design rationale of Ten15 is to provide an algebraic basis for software development. This leads to a mechanism by which different computers can be made compatible through the implementation of the Ten15 algebra on each platform. In addition, Ten15 is a complete systems programming language, permitting the fine-grained allocation of mainstore, filestore and network resources. The facilities of the language extend its application area to fields such as high integrity systems, secure systems, IPSE development, heterogeneous networks and fine-grained databases. This is associated with a mathematically described strong type system, which ensures the type security of the system functions. The use of Ten15 means that its users can preserve their investment in existing software because Ten15 can co-exist along-side existing operating systems and, since it is an intermediate language for the majority of modern programming languages, such as Ada, Pascal, and Standard ML, it can provide a mechanism for mixed-language working. With regards to database development, the salient features of Ten15 are that it possesses a mature type system which guarantees the type integrity of programs developed using the Ten15 notation, an immutable persistence mechanism, and that it has a model of concurrency and transactions.

3.1. Persistence

Ten15 incorporates a persistent filestore, based on a mechanism used in the KeepSake database kernel [32]. The persistent filestore is called the *datastore*, into which arbitrarily complex Ten15 data fragments can be persisted between program executions. The datastore is novel in that it is immutable, or non-overwritable. The datastore is structured hierarchically, radiating down from a single root, which is the only location that is overwritable. The root is protected by a four block, three stage commit algorithm to preserve its integrity in the face of error or corruption. There are several advantages to immutable filestores over the more traditional overwritable ones. First, it makes the filestore more resilient to corruption from a premature commitment due to a crash or an error. This is because, until the root is assigned with a reference to the new state of the datastore, the old database is valid. If a crash occurs whilst the root is being written, the subsequent system initialisation will determine whether the update operation on the root was completed, since the root is not assigned unless the three protection blocks contain the same value. A second advantage is that the history of changes to the filestore since the last garbage collection is preserved in the immutable store, allowing privileged software to undo the changes to persistent data structures. Thirdly, it provides a many readers, single writer mechanism for concurrent update, since any number of processes can access the original root while a process is preparing a modification ready for assignment to the root.

Data is explicitly preserved in the datastore by using a Persist operator provided within the Ten15 notation. When Persist is invoked on an arbitrary piece of data, the data is compacted into a persistable format by tracing and concatenating all objects to which the data refers, altering all pointers and offsets within the concatenated block to reflect their new status as datastore objects, and writing the block onto a new location in the data store. A reverse operation, UnPersist, takes a persistent reference and delivers the mainstore representation of the persisted value, by copying the persistent block back into mainstore, updating the pointers within the block to their new absolute location in mainstore. This approach to persistence is an example of *replicating persistence* [9], since repeated unpersisting of the same persistent object causes two identical copies of the object to be brought into mainstore.

Persist and UnPersist provide a means of writing and reading a value to and from persistent storage. Due to the nature of the immutable storage, updating of a persistent value cannot occur in place. As a result of this, the Ten15 datastore supports *persistent variables* (pvars) which control the updating of single instances of persistent values. The mechanism for managing updates to persistent variables is to create a copy of the datastore by updating the root of the datastore to refer to a new datastore where the persistent variable has been updated to reflect the change in the persistent value, and the hierarchical structure of the datastore has been altered to propagate the changes.

Persistent variables provide a means by which object identity of persistent objects can be preserved. The pvar acts as an outwardly unchanging reference to an object on the persistent store, since changes to the pvar are propagated to all references. Referential integrity is preserved by a process of path following of updates to the pvar, with the final update being the current value for the pvar object.

A hierarchical structuring mechanism can be used within a Ten15 datastore. The data structure for this is the *persistent set* (pset). If a datastore were flat, eventually as more pvars are added to the datastore, the data structure handling the pvars would become unwieldly and inefficient. Hence Ten15 supplies a mechanism for decomposing pvars into a tree of logically related pvars and psets, where only the group of pvars and all the groups of psets and pvars directly leading to the root of the datastore are updated when a pvar is reassigned. This is displayed in figure one, where a persistent value is updated and a hierarchical structure of pvars and psets is updated as necessary to reflect the new nature of the datastore.

Since the datastore is constantly being written to and no parts of it are ever re-used, there comes a time when the datastore becomes full. As a result of this, a garbage collection algorithm is called which clears away all persistent values and variables that cannot be traced from the root of the datastore. A similar event must take place in the mainstore to clear out all data that can no longer be reached from the existing execution display. An efficient garbage collector for both the mainstore and datastore is central to the success of Ten15, since both resources are hidden from the programmer s/he cannot exercise discretion in allocation in order to avoid the need for the release of memory or datastore that are no longer required.

The distinguishing feature of the Ten15 persistence mechanism in comparison to other

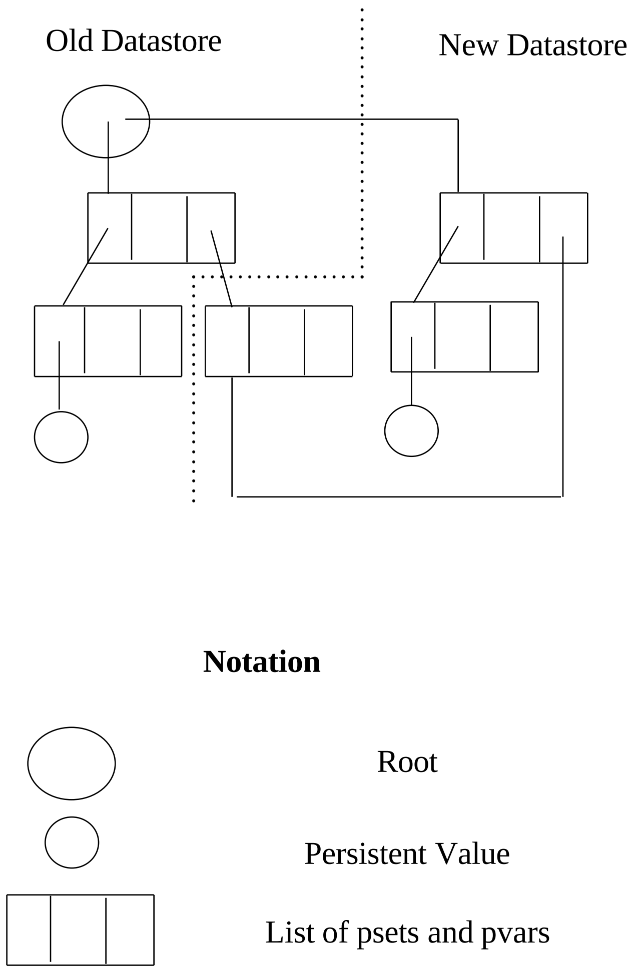

Figure 1: Updating a persistent value in a datastore.

PPLs, such as Napier88 and PS-algol, is that it was initially developed as part of an investigation into type systems and compiler construction. As a result of this, the persistence mechanism came mainly as an afterthought. The persistence mechanism is the minimum necessary to adequately support the arbitrary persistence of data. In addition, the form of persistence exhibited by PS-algol and Napier88 differs from the Ten15 persistence mechanism. PS-algol and Napier88 have a persistence mechanism based on *environments*, which are infinite unions of all labelled cross products [20]. They are tuples of name to complex value mappings, which can be incrementally and dynamically built up and reduced. Ten15 uses a value based persistence mechanism [28], where a capability to the persistent value is the only form of reference. The naming approach introduces a degree of complexity into the underlying implementation which supports persistence, since the uniqueness of names must be guaranteed and comparison of names must be made at run-time. The value based approach means that the supporting software is simpler since the persistent capability is essentially a disk address, a representation of the type of the persistent data and an indication of the size of the stored data. The capabilities can be quoted freely within data structures in the language, without any run-time overhead for the evaluation of names, and, of course, uniqueness is totally guaranteed. On the minus side, the capability approach means a greater complexity in the development of the supporting software since capabilities must be unique, secure, unforgeable and able to address a possibly large address space, especially in a distributed system, where remote access is incorporated into the capability.

3.2. Type system

Ten15, like Napier88, incorporates a number of modern type system features, such as lambda type functions, existential quantification, extended union, dynamic typing and first class closure [15]. Ten15 has a strong type system, in that the type of every value and expression is resolved at compile time. Type equivalence is of a typed structural equality basis, such that two values are equivalent if legal typed selectors can be applied to either to give the same result [4].

The Ten15 type system provides a basis for the fulfillment of the criteria laid out in section two. Each of the features is now analysed in turn to demonstrate their efficacy in database data structure modelling.

Lambda type functions – A type function is the equivalent of a parameterised type. In Ten15, a polymorphic type can be declared as an arbitrarily complex collection of type constructors and polymorphic values declared at compile time. This is demonstrated in figure two, where a lambda type is declared which creates a vector of type ptr X hidden behind an abstraction called Set. Ptr X is a declaration of a normalised polymorphic value in Ten15 where the normalisation is achieved by referring to the polymorphic value through a pointer type which is of a known size. The lambda type can be instantiated as a particular type, such as a set of pointers to positive integers. The relevance of this feature to database structure modelling is that an abstract concept, such as a set, can have a representation hidden from the database applications programmer. Other PPLs support parameterised types, but Ten15 allows arbitrarily complex combinations of modes (Ten15 types) to be hidden within the abstraction, through the use of such type constructors as structures, unions, cycle (recursive types) and vectors.

```
PosInt = range(0, Maximum_integer);
Set = λ(X: vec(ptr X, PosInt))

Used as: Set[X], Set[PosInt] ....
```

Figure 2: A lambda type function.

Existential quantification – Existential quantification provides a mechanism for controlling what is known about an object. An existential value is essentially a mechanism for hiding the type of a value, until a means exists to access the internals of the existential using a procedure called *skolemization*, or by providing a key. Figure three is an example of a Ten15 existential value which contains a polymorphic data object and a first class procedure which reveals information concerning the data. The existential Obj can be instantiated to any type, and every instance of Obj will be type equivalent. However, once the existential object has been skolemized, information concerning the data held within it can only be discovered by applying the first class procedure within the existential to the data. The example is also a simple definition for an abstract data type, and it is for this reason that existentials are useful for database structure modelling since they provide a mechanism for the aggregation of data and objects into a format suitable for persistent storage and a way of creating an object with the necessary amount of information hiding [5].

Abstract data types are provided for in other PPLs, notably Napier88 with its **abstypes**, and Galileo.

```
Obj = ∃(X: struct(ptr X, (ptr X → Int)))

Let addone = Proc (i : ptr Int) -> Int: i + 1 Endproc
(* procedure adds one to an integer *)
Let packed_int : ptr Int = Pack CrInt 5
Let int_object : Obj = ToExists: struct(Obj, Int){(packed_int,addone)}
(* create an existential object instantiated to type integer *)
Let (value,proc) = Skolemize int_object
(* skolemizes integer object to reveal contents *)
In (* applys addone procedure to value - result is an integer of value 6,
     otherwise program fails if existential was not
     instantiated to type Int *)
  As: Int {proc(value) | Fail}
Ni
```

Figure 3: An existential quantification over an integer ADT.

Existentials provide a simplified means for providing dynamic typing in Ten15. When the existential is generated, an unique value of the instantiated type is stored with it. When a skolemization takes place, the fields of the resulting structure are universally quantified with the type representation. When fields are subsequently reference or applied to one another, the resulting type is the least upper bound of the application type and the expected result type.

Extended unions – Extended unions are a refinement of the Ten15 mechanism for existential quantification. They provide two useful properties: security and dynamic typing. An extended union is effectively a value and unique pair hidden within an existential abstraction. Ten15 has a conditional skolemize operator, **Match**, which skolemizes the existential pair if a unique key value is provided as an argument to the operator. The unique key value has to be the same as the one provided at construction time. Since unique keys are unique system wide and cannot be forged due to the strongly typed nature of the Ten15 type system, a capability based mechanism for mainstore security can be provided. The second property, dynamic typing, as discussed later, is a result of the way that unique keys are generated. They are effectively a unique integer paired with an unforgeable numeric representation of the required type. Hence when a Match is made against a key and a union, the type of the result can be predicted and hidden information used in a type safe way. Figure four is an example of using an extended union. Unique defines the particular type of an integer. An extended union is instantiated with the integer type and an integer. An assertion is made as to the internal type of the extended union. If the two type capabilities are the same, then the contents of the extended union are revealed, otherwise a type mis-match failure is generated. Reference [11] contains a more detailed account of using existentials for security purposes.

Extended unions are suitable for database programming for two reasons. First, they can provide a language based mechanism for object security. Secondly, since unions are

```
Extended_union = ∃(X : struct(unique X, ptr X))

Let unique = MakeUniqueInt ()
(* generates a unique value of type unique Integer *)
Let ext = ToExists:struct(Extended_union, Int){(unique, Pack CrInt 5)}
In
   As: Int Match{(ext, unique)| Fail}
   (* assertion yields 5 or fails if the uniques do not match *)
Ni
```

Figure 4: Using an existential as an extended union.

based on existentials, it is possible to provide a unified mechanism for the storage of data structures of a mixed type, using first class closures to tailor access and update operations to the types of particular elements.

Extended unions can be found in other PPLs. The environments of Napier88 are persistent versions of vectors of name and extended unions pairs. A name is bound to a particular union, with the value bound to a particular type. When the value is to be referenced, an assertion is made which checks that the type of the value is the expected result type.

First class closure – First class closure provides a means of delaying the binding of parameters to procedures until run-time and it allows for the partial application of procedures. This is achieved by issuing a Close command at run-time to a procedure and the necessary parameters. Closures may be nested to any level. Figure five is an example of the first class closure of an upper bound for a vector delayed until run-time. A non-closed procedure (*Ion*) is declared which expects to be bound with a non-local (size) before it can be executed. The binding of non-locals to ions is performed by the Close operator.

```
Let test =
          Ion
            (size : Int)
              (value : Int)
                 → vec(Int, Int):

            Vec(size Of value)
          Endion
In
   (CrInt 5 Close test) (CrInt 15)
Ni
```

Figure 5: First class closure in Ten15.

This example shows us the value of first class closure for database programming is that the internal characteristics of the system need not be determined until run-time, allowing for greater generality. In addition, closure can be used to limit the scope of the polymorphism provided by a procedure by insisting that first class procedures defining operations on the polymorphic type are closed with the procedure before it can be executed. First class closure can be simulated in persistent languages such as Napier88 and Quest [1, 14], using higher-order first class functions, and they are automatically provided for in lazy functional languages.

Dynamic typing – Dynamic typing is the ability for a data value to carry its type with it at run-time. This is effectively achieved by pairing the value with a representation which always denotes values of that type and cannot be forged. The Ten15 type mechanism is implemented in this way, where a value is associated with a mode (type) which uniquely defines the type. Using the least upper bound for modes rule, whereby the numeric representation of a type can be coerced to other similar modes, such as characters to integers, values which have an equivalent least upper bound for their modes can be treated as values of the same type [25]. As with extended unions, dynamic typing is useful for representing values of a mixed type within the context of a single data structure.

A combination of these features means that Ten15 has the expressive power on which to model database structures and support a model rich enough to allow the creation of objects. This is demonstrated in the following example, where a tuple data type is defined which provided efficient access to keys in large tuples.

A tuple can be implemented using an extensible hashing algorithm, provided that the key value can be hashed onto an integer. This can be defined as the Ten15 lambda type function:

```
PAIR = λ(X : struct(STRING, ptr X))
PAGE = λ(Y : vec(PAIR[Y], PosInt))
TUPLE = λ(Z : cycle(vec(union(TUPLE[Z], PAGE[Z])),PosInt))
```

This creates a cyclic data type which either refers to another level of the tuple structure (vec(TUPLE...)) or a page of name to polymorphic value mappings (vec(PAGE...)). To instantiate the tuple data type and to allow a tuple to have mixed type, the tuple must be bound to an extended union data type, called affectionately Wotsits.

```
let aTuple : TUPLE[Wotsit] = ....
```

An abstract data type is needed to define the operations permissible on the tuples. Since extended unions require that the mode representation for a value is not defined until run-time, first class closure is necessary to allow a procedure to be bound with the mode of the particular field of the tuple. The ion indexTuple expects the closure of a unique value which defines the type of a particular field of the tuple. Once this is done, the field can be indexed and the result delivered provided that the closed mode matches the type of the extended union for that particular field. In the example, Formals X denotes a polymorphic value is used within the ion.

```
let indexTuple =
    Ion Formals X
      (mode : unique X)
        (aTuple : TUPLE[Wotsit], key : String)
           → ptr X:
    ....
    Endion
```

Existential quantification can be used to define the abstract data type for tuples, and provide information hiding of the representation of the tuple.

```
TUPLE_ADT =
  ∃(X :
     struct(
          TUPLE[Wotsit],
          (unique X → (struct(TUPLE[Wotsit],String) → ptr X))},
          ....
     )
    )

tuple = ToExist: struct(TUPLE_ADT, X){(aTuple, indexTuple, ...)}
```

Using the advanced features of the Ten15 type system, complex database abstractions can be represented and provide the applications programmer with a seamless database programming environment.

3.3. Concurrency

The Ten15 abstract machine has a well defined model for concurrency , inter-process communication, transactions and remote indirections [11].

Concurrent processes are achieved through the launching of first class procedures bound to a set of parameters. Synchronisation and communication takes the form of typed queues. The sender places a typed piece of data into the queue, which the receiver either waits for, thereby providing synchronous communication, or times out should the data not be delivered within a certain time, thereby providing asynchronous communication.

Within the abstract model that defines Ten15, there is a notion of transactions. Ten15 transactions are a data type which are incrementally built up out of persistent variables which must be committed at the same time, to avoid inconsistencies in the datastore. As with the traditional notion of transactions, the completed transaction can be committed or abandoned without affecting the datastore. A number of processes can build up transactions in parallel and, due to the flexible model for transactions adopted by Ten15, it is theoretically possible for deadlock to occur if shared pvars are added to two or more transactions in differing orders. However, Ten15 creates transactions based on a canonical ordering of pvars before any attempt is made to insert them. Thus, in the case where two transactions which are to share two common pvars are created in parallel, the possibility of deadlock is eliminated.

The Ten15 notion of persistent reference and variables provides an effective means of abstraction over placement of persistent data across several distributed datastores. Since the parallelism model is hidden beneath the notation, it is possible for local placement, remote placement and object migration [27] strategies to be implemented without affecting Ten15 code written for a single node. A problem with this approach is that there is a loss of efficiency in the resolution of capabilities and the some of the flexibility of placement is taken away from the programmer.

In conclusion, although concurrency is not available in the Ten15 notation, a model exists on which a conceptual distributed database management system can be implemented. Transactions are available on the Vax implementation of Ten15 so that the update of a set of pvars can be committed through the single writing of the root.

4. Probe

In this section the Probe object database system is presented. The architecture underlying Probe is introduced and analysed, and mention is made of its salient features. In addition, we demonstrate how Probe is an attempt to solve the problems of databases developed using persistent programming languages, as discussed in the introduction. We pay particular attention to the programming language TDBPL which is our attempt to provide a coherent notational semantics for the Probe abstract database machine.

4.1. The Probe architecture

Figure six shows the levels that comprise the Probe object management system.

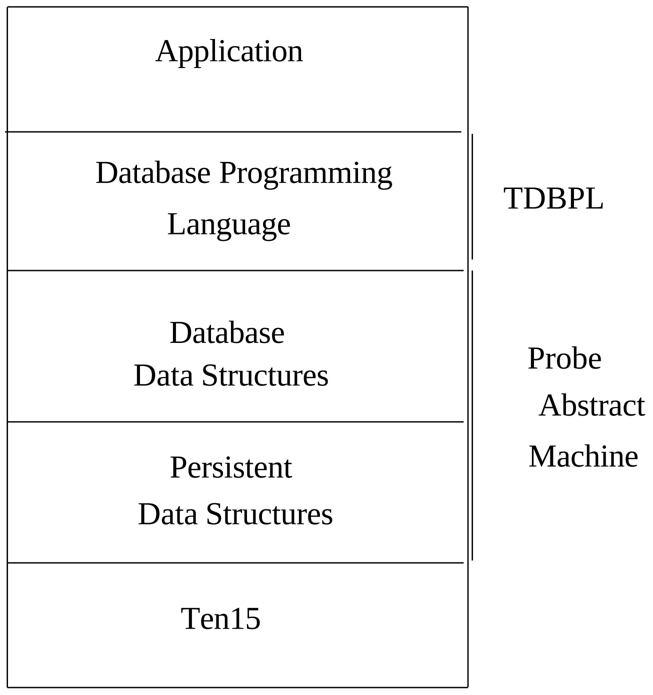

Figure 6: The Probe architecture.

4.1.1. Ten15

Ten15 provides the development language for Probe. Hence Probe is developed in a type-secure environment, with a value based persistence mechanism and concurrency control using transactions. Ten15 permits buffer, mainstore and datastore management, datastore security and data integrity.

4.1.2. Persistent data structures

This level provides data structural abstractions over the persistent store. The philosophy in their implementation is to hide the persistent store beneath a level of well-defined abstractions, such that the locality of a value is invisible to the programmer. In addition, the level should allow access and update of values with the minimum number of interactions with the persistent store, and with the minimum amount of data having to be brought into mainstore to achieve this. The mechanisms available in this level are:

- *Accelerators*, which are a means of providing fast access to bulk data through the manipulation of persistent references. The indexes to bulk data make use of extensible hash tables [23], B-trees [18] and fixed indexes. The approach for the implementation of the accelerators is to decompose the data structures which fit into chunks which can be retrieved in one backing store access (BSA). In the case of extensible hash tables, this approach is highlighted by the decomposition of the hash table into nested chunks of 'bit slices' of the hash key. Also, another goal is to provide generality of keyed access, so that an arbitrary data type is as valid a key as a string, provided that there is a suitable hashing algorithm for the data structure. Generality allows for a larger application domain for data structures of the level.
- *Associative matchers*, which map names onto persistent values. The data structures permit related polymorphic keys to be mapped onto values to be represented on the persistent store. They are used in association with the extensible hashing accelerator to provide storage for name to value pairs.
- *Sequences*, which group together values of similar characteristics, locality or type. These are similar to associative matchers, expecting that values are stored using a numeric index as the key.
- *Compactors*, which provide compact representations for persistent data that have similar values. For example, sequences in which every index has an identical value can be represented as a single value and a pair of the sequences upper and lower bounds.

4.1.3. Database data structures

The database data structures make use of the data structures defined in the persistence level, to provide the abstract data types that make up an object database model. The implementation policy for the model is to minimise the overhead of frequent access to the persistent store by providing a mechanism whereby the database data structures can be partially built up and evaluated in mainstore. This necessitates that for changes to be propagated to the store, an explicit commit operation must be applied to the data structure. Failure to commit a data structure is equivalent to aborting the partial transaction. Used in conjunction with Transactions, this means that no unnecessary writing to disk needs to be made.

Transactions on database objects must be partial due to nature of Ten15's memory organisation. Values are kept in memory until there is no longer a reference to the value from any active data structure. In the context of database objects, it is possible that a data structure will be larger than mainstore, and since it will be active, none of its limbs can be cleared out by the garbage collector, and so the memory will become full. Two solutions exist to this solution: first, the *brute force* approach is to commit the entire structure to the datastore, and start again with the new copy, since each limb will now be replaced by a persistent value rather than the mainstore expansion of the data structure. A second approach relies on an intelligent commit algorithm. Each data structure is allocated a maximum partition size. As a data structure is incrementally brought into memory, it will grow beyond the bounds of the partition. When memory becomes tight, it is possible to apply the partial commit algorithm which will persist limbs of the data structure, replacing expansions with persistent references, until the data structure is back within the bounds of its allotted partition. The approach is analogous to providing a paging policy for each database data structure. When the partial commit is invoked is left to the discretion of the applications programmer. The object is not fully committed until the pvar is assigned with the new persistent reference.

It is currently a matter of research as to the optimum mainstore representation to provide a suitable basis for the implementation of the partial committal algorithm. Preliminary investigation suggests that the mainstore representation should consist of a nucleus, which extends to the boundary of the partition size, consisting of the roots to any persistent data structure, key information, such as the size and bounds of the data structure, and any hot spots, i.e. values that require optimized access since they are frequently referred to. The concept of hot spots is discussed in section 4.3.

Each data structure in the database level has three things in common. First, they can have three variables types: a main memory representation, which can be used for the incremental building of the data structure, or as an intermediate store for results and inter-transaction communication; a persistent representation, which permits the data structure to be represented on the persistent store as a value; and the persistent variable representation, which allows the data structure to preserve object identity. The main memory representation can be derived from the other two forms by incrementally loading those persistent values that are referenced, so that values within the data structure can be changed without having to repeatedly read or write from or to disk respectively.

Second, each data structure allows the specification of mainstore and backing store constraints. When a data structure is created, it expects the closure of a persistent set, which specifies the location of the object in the datastore hierarchy, the size of a value in bytes and the size of a disk block, so that levels of the accelerator indexes can be fitted into a single backing store access. In addition, a cluster size is required, which defines the optimum size of the core data structure that is required to remain in memory.

Thirdly, each data structure allows for *persistent mutation* whereby it changes to its optimum, most efficient representation over its lifetime, based on access and writing trends. This concept is discussed more fully in section 4.3.

The data structural abstractions of the object database model include sets, tuples and databases.

- **Tuple** A tuple is a fixed size set of string to extended union pairs, where each extended union is an instantiated polymorphic value. The tuple is stored as a fixed index, with the name field mapped onto an index using rehashing. For large tuples, i.e. large than one BSA, the index is decomposed into blocks. The leaves of the index are pages of keys to value pairs. Operations on specific keys in the tuple require the closure of a unique value which represents the type of the resulting value. If the type of the closed value and the extended union match then the required value is returned.
- **Uset** A Uset (unordered set) is a set of values stored in an unstructured manner. Values must be unique, requiring the closure of an equivalence operation. This form of set is inefficient for large sets, since operations such as to test for membership or addition require iteration through the entire set to test for uniqueness. Usets are implemented on a refinement of the extensible hashing data structure, where each value is assigned an unique numeric key when it is added to the uset.
- **Oset** An Oset (ordered set) is a set of values stored with a partial ordering, the partial ordering being based on the provision of a hashing operation for the stored value. Osets bring efficiency to the storage of large sets since tests for membership and addition requires the resolution of the hash of the value to be stored. If the hash key is not unique then pages of values can be created. Should a page become full, a simple rehashing

algorithm can be used.

- **Database** The Database type allows for the coexistence of a single database in the process of change. The change may be the development of the schema describing the structure of the database. This data structure is discussed more fully in the next section.

In addition, this level supports atomic data types composed using the persistent data structures. Atomic data types are bulk constructors for data types that do not conform to a particular data model, i.e. the data structure cannot be expressed in terms of the data structures provided by a database model. Vectors, arrays, queues, lists and stacks are examples of atomic data types. However, atomic types can make use of the same data structures to provide efficient persistent representations. Like the data structures defining the object model, the atomic data structures can have mainstore and persistent representation, depending upon their respective needs to be persistent which is the responsibility of the applications programmer.

The database data structure level also supports statistical analysis, so that optimization on the location of particular values in the persistent store can be performed independently of the persistence level data structures. This is one of the more important features of Probe, which is discussed fully in the next section.

4.1.4. Ten15 Database Programming Language

Probe provides the basic mechanisms to support an object database model. In this section we discuss a concrete programming notation that resides on Probe, eponymously called the Ten15 Database Programming Language or TDBPL. The rationale behind the production of TDBPL is discussed in conjunction with examples demonstrating the problems arising in using Probe as a basic model for database construction. The equivalent examples are then present in the TDBPL notation in order to demonstrate the expressiveness of the language.

4.1.4.1. Why yet another programming language?

As discussed in the introduction, DBPLs combine the facilities to build a database with the expressive features of a programming language from a particular programming paradigm. We have three main criteria for a DBPL, derived from the experiences of reference [13]:

Data model – The DBPL should have a powerful modelling ability, so that any reasonable database model can be constructed within the language, In addition, the data model should provide orthogonal storage capabilities for data structures that are external to the model yet require an efficient representation (*atoms*).

Expressive power – The DBPL should possess a rich variety of operations and data structures, so that the data model can easily be combined and reasoned with by operations that are independent of the database.

Underlying efficiency – The DBPL should be constructed upon an efficient database kernel, so that the database application programmer does not have to ‘break’ the model to provide maximum efficiency for a database system. Garbage collection and data reorganisation should be part of the database kernel, so that the representation of a database on backing store is hidden from the programmer and cannot be altered.

Probe satisfies all three criteria. First, Probe supports an object based model, allowing complex objects to be arbitrarily nested, using sets and tuples. In addition, the entities of the database can be drawn from the infinite number of data structures that can be constructed using the Ten15 type constructors, or from the set of *persistent atoms* defined in Probe, which provide efficient persistent representations for traditional programming language bulk constructors, such as fixed vectors (SVEC), dynamic vectors (DVEC) and arrays.

Second, Ten15 has a set of powerful operations and control flow facilities. These can be combined with the data retrieved from the database and the database operations to improve the expressiveness of the language.

Third, Probe provides an efficient platform for database development, due to its reliance on optimised accelerator data structures to give access to data. Data re-organisation is undertaken on an heuristic basis, with analysis of operations on data taken as the basis for a policy of data structure mutation.

As an abstract machine, Probe possesses the three desirable features of a DBPL. However, there are several features of the machine that are not consistent with a coherent DBPL, and it is therefore prudent to layer a notation on top of Probe. The inconsistencies of Probe are:

- Due to the way in which attributes of a TUPLE type are constructed, the type representations of specific fields are instantiated at run-time, and are unique for that particular execution. Hence two databases with the same schema and containing identical values can be type-wise incompatible if they are created at different execution times. In providing a notation, a degree of consistency can be applied to type representations, in that a particular type is incorporated into the compiler. When a new type is defined, this is dynamically added to the compiler data structures for future reference.

- The data model level of Probe relies on the closure of several values which describe the underlying structure of the persistent heap and the machine architecture, such as a polymorphic hashing function, the size of data stored, the optimum size for blocks and clusters of blocks. The inclusion of the values detracts from the readability of the schema definition, and the values can be standardised for a particular compiler, or derived from compile time analysis of the data structures involved.

- Each data structure in Probe can be memory resident, read only, or can have both read and write access capabilities, which can be arbitrarily applied in defining a schema. By hiding the decision process under a notation, the choice of whether a value should be read only, or readable and writeable, or whether the value should be kept in memory whilst intermediate operations are applied to the data structure, can be based on heuristics in the compiler and static analysis.

- Probe provides a number of disparate database data structures. Each is logically complete, in that update operations are either committed or are aborted, without leaving the data structure, or the datastore, in an inconsistent state. However, a database is typically a nested set of the data structures, and so update operations must affect the entire database, or leave it unchanged. As a result, update operations on individual data structures must be coordinated through the use of transactions. The management and composition of transactions can be made a part of a notation compiler, since the compiler will have a greater understanding of the structure of the database and so be able to allocate transactions accordingly.

- Since key fields of tuples can be identified at compile time, it is possible to provide optimisations in retrieving values that use the key value field as part of the query. The language should be able to determine when a key value is being accessed and substitute a specific index operation instead of a general iteration to find the value.

- Probe lacks a graceful exception handler, in that Ten15 insists that if protection from exceptions is required, procedure invocation must be made conditional upon the likelihood of failure†. This hinders the programming style of Probe and the necessary code could be generated automatically by a compiler of the notation.

- The final justification for a notation for Probe is more pragmatic, in that the notation that Ten15 uses is, although expressive, intellectually difficult to reason and program with. A new notation would hide the less intuitive parts of the language under ‘syntactic sugar’.

These limitations have guided the definition of TDBPL.

4.1.4.2. Programming a database application using Probe

This section describes an actual database built using Probe, highlighting the notational inadequacies of the abstract database machine. Due to the limitation of space, this section provides only brief demonstration of using the Probe system as a development environment for database applications.

The application developed in Probe is a simple employee database, where an employee is represented as a tuple, consisting of a name, which acts as a primary key, an age attribute and a salary. Employees are collected into an ordered set, using the uniqueness of the name to provide a partial ordering, dependent on the hashing algorithm applied to the key field.

The schema for the the employee database is represented using Ten15 modes (types) as:

```
Employee = TUPLE
Database = OSET[TUPLE]
```

Figure 7: The modes defining an employee database.

The schema defined in Figure 7 contains little information about the nature of the employee database. Tuples are created dynamically, and every tuples has the same basic type, hence there is no provision at compile time for the definition of attributes of the tuple object. A Probe database schema is effectively an informal collection of types, whose interactions and constraints are defined dynamically. Hence the schema makes no provision for the integrity of the database application, nor does it define the specifc details of the application, such as attributes and ordering policies.

Tuples are created dynamically, using the Probe operations *crtuple* (create tuple) and *addtuple* (add tuple), which require the late binding using first class closure of the characteristics of the underlying datastore, such as the size of data stored and the size of a backing store block, and the placement of the tuple in the persistent store hierarchy. This is demonstrated in Figure 8*. The code in Figure 8 makes use of the database types defined in Figure 7. The Employee type is returned from the create_employee operation since the operations on tuples

† The Ten15 operator *Trapply* is used to conditionally apply a set of parameters to a function. An alternative branch of code is supplied to deal with a failure in the procedure application. The following example demonstrates, where 4 is applied to a function which adds one to an integer. In the event of a failure, such as overflow, then the second limb (|) of the result code is subsequently called.

```
Trapply {(4),add_one | Op x: writei(x) | write("Failure")}
```

return a structural equivalent type to the Employee type.

```
Let hashstring = Proc (str : String) → PosInt: {body omitted} Endproc
Let eqstring =
        Proc (str1 : String,str2 : String)
            → bool: {body omitted} Endproc

Let create_employee =
        Proc (name : String, age : PosInt, salary : PosInt)
            → Employee:

  Let fields = ["Name","Age","Salary"] {simplified vector constructor}
  Let initial_tuple =
     ((16, 512) Close (stringmode Close (pset Close crtuple)))
       (3, name, fields, RW)
  { crtuple defines the outline of the tuple with 3 fields,
     assigning the first field of the vector fields with name.
     (16, ..) is the data size and the block size of the underlying
     machine. Stringmode defines the type of the Name field.
     Pset defines the placement of the tuple in the datastore
     RW makes the tuple readable and writeable}
  Let partial_tuple = (posintmode Close addtuple)
                    (initial_tuple, age, "Age", MEM)
  Let complete_tuple = (posintmode Close addtuple)
                    (partial_tuple,salary, "Salary",MEM)
  { addtuple incrementally builds up the tuple in memory (MEM), assigning
     each key of the tuple with a type and a value of that type}
  In
    commit(inital_tuple,complete_tuple)
    { commits RW tuple with complete mainstore tuple }
  Ni
Endproc
```

Figure 8: Creating an employee tuple.

By invoking the *create_employee* procedure defined in Figure 8, the resulting employee can be used to create an employee database. Figure 9 demonstrates the creation of an employee database in this way. The ordered set (OSET) data type provides a partial ordering of its elements through hashing. With respect to the employee type, a hash function is defined which returns a hashed value based on the primary key of the value. In Figure 9, a database is created using *croset* (create oset), and tuples are subsequently added to it using *addoset* (add to oset). This links in with the types defined in Figure 7, since a database is defined as an OSET of employees. *Create_employee* delivers a value of type Employee which can be added to the

∗ For the sake of clarity, and due to the obliqueness of the Ten15 type system, the notation has been somewhat simplified in the all the examples given.

OSET created in *new_database*.

```
Let eqemployee = Proc(e1 : Employee, e2 : Employee)
                 → bool: {body omitted} Endproc
Let hashemployee =
       Proc (e : Employee) → PosInt:
           hashstring(
               (stringmode Close indtuple)(e,"Name")
             )
         { Indtuple returns the value associated with the specific
             key value with the closed type. In this case, the primary
             key name field is return so that it can be hashed.}
       Endproc
Let new_database =
       Proc () → Database:
           Let tuple1 = create_employee ("Smith",23,25000)
           Let tuple2 = create_employee ("Jones",45,18000)
           Let database =
              (hashemployee Close ((64,512) Close (pset Close croset)))
                   (tuple1, RW)
              { croset requires the closure of the hashing function for
                  employees, the size of data to be stored and a location
                  in the persistent hierarchy. As parameters, it takes a
                  tuple of type employee and a read/write capability.}
           In
              (eqemployee Close (hashemployee Close addoset))(database, tuple2)
              { addoset adds an employee to the database, using eqemployee
                  to check that tuple is unique, and hashemployee to
                  specify a lcoation for storage of the value.
           Ni
         Endproc
```

Figure 9: Creating a database of employees.

Queries on values held in the employee database take the form of path following. For example, a query such as ‘What is Smith’s salary?’ will result in every element of the set being searched until an employee called Smith is found, whose salary is then returned. Probe allows iteration through sets using a *for_each* operator. The *for_each* function iterates through each value in a set and performs a programmer-supplied action on the value. If the result status of the action is satisfied then the iteration terminates. This is demonstrated in Figure 10, where a version of the query defined above is highlighted.

```
Let db = new_database()  {defined in Figure 9}
Let introduction : PosInt = CrPosInt 0 {value will instantiate action}
Let action =
    Ion
     (name : String)
       (e : employee, result : PosInt) → (bool,PosInt):
            {action requires late binding of name}

              Let tuple_name = (stringmode Close indtuple)(e,"Name")
              {get name from this employee}
              In
                If name = tuple_name
                Then {found the right person → deliver salary and cease searching}
                  (True,
                    (posintmode Close indtuple)(e,"Salary"))
                Else {not found → continue looking}
                  (False, result)
                Fi
              Ni
            Endion
        Let smith_salary : PosInt = for_each(introduction, ("Smith" Close action), db)
```

Figure 10: What is Smith’s salary?

The example in Figure 10 demonstrates the general form for retrieving an arbitrary attribute from a set of tuples. However, it is possible to provide an optimisation for the operation by exploiting the way in which values are stored in a partial ordering in the set. In the example database, the Name field is used as the basis for the hash function which is closed into all the operations on the set of employee. The query ‘What is Smith’s Salary’ makes use of the Name field to ascertain Smith’s salary. The weak type of tuples and the operator *findoset* which takes a set and a value and returns the equivalent value stored in the set can be used to provide an optimised access for a specific name field.

Figure 11 is an efficient rewriting of the query solution of Figure 11. Since hashing and equivalence testing of employees rely solely on the Name field, a minimal tuple, with the required Name field as its only attribute, can be given to findoset, and the operation will return the equivalent tuple with the additional fields.

```
Let partial_tuple =
        ((16, 512) Close (stringmode Close (pset Close crtuple)))
          (1, "Smith", ["Name"], MEM)
Let full_tuple = (eqemployee Close (hashemployee Close findoset))
                (database, partial_tuple)
Let salary = (posintmode Close indtuple)(full_tuple,"Salary")
```

Figure 11: An efficient implementation of the ‘What is Smith’s Salary’ query.

The examples in Figures 7-11 demonstrate that Probe provides a logically complete DBPL. It provides a fully functional object database model, using the tuple and set primitives. It possesses the ability to finely tune the storage policy for database data structures using the late binding of the size of data and backing store blocks, and of the specification of the attributes that are be used as index values. It possesses expressive power since Probe and Ten15 data types and operators can be combined in an orthogonal manner. Efficient storage of values stored using the database constructors is hidden beneath the abstraction. Reliable and resiliant storage is provided for by the underlying Ten15 persistent storage implementation.

Probe conforms to the three criteria for a DBPL described in section 4.1. However, the notation Probe uses is obviously awkward, and the meaning of operations is not readily apparent, and that the structure of database is open for destructive operations, since the schema provides only a general description of the databae and does not enforce integrity.

4.1.4.3. Programming using TDBPL

TDBPL is an attempt to alleviate the problems of lack of clarity of Probe. It achieves this through providing a more succinct syntax for operations and relying on inference to form complete Probe operations.

Before introducing TDBPL, it is worth discussing the rationale of layering TDBPL on Probe, as opposed to the approach of implementing TDBPL as an independent DBPL. There are a number of advantages and precedents. First, it compiling down to a high level language is easier, from an intellectual and software engineering viewpoint, than to machine code. Secondly, Probe provides an existing, architecturally neutral, efficient set of database primitives which are combined with the very expressive Ten15 programming language. Thirdly, the approach for building PPLs has been to provide a persistent architecture beneath the programming language. Examples of this approach include POMS [16] and PAIL [12, 21] which are the persistent object stores underlying implementations of PS-algol. The mapping between Probe and TDBPL is the database programming language equivalent of this approach.

Figure 7 gave a Probe schema definition for an Employee database. A problem with the definition is that all values of type TUPLE are structurally equivalent. TDBPL can apply name equivalence to tuples by generating more type information from an expanded schema definition, as demonstrated in Figure 12. TDBPL can now create a type representation based on the extra information given with the tuple, which is paired with the tuple, and used to prevent inappropriate tuples being applied to procedures. The textual description of the schema can be used as a basis for an auxiliary data dictionary data structure. When an operation is applied on the tuple, the data dictionary is checked to ensure that the key value is valid. The addition of the *Key Field* keyword permits the automatic generation of hashing and equivalence functions, which must be closed into ordered sets at the Probe level. If the Key Field keyword is omitted then it is left to the discretion of the language to decide on a hash key field, chosen at compile time or altered dynamically using statistical analysis of the data structure.

```
Employee = TUPLE [Name : Key Field String, Age : PosInt, Salary : PosInt]
Database = SET of Employee
```

Figure 12: A schema in TDBPL.

The task of building database objects can be hidden beneath the syntax of TDBPL. Figures 8 and 9 demonstrate the construction of an employee and a database respectively. Figure 13 is the TDBPL equivalent of the two functions.

```
e1 : Employee = ("Smith",23,25000)
e2 : Employee = ("Jones",45,18000)
```

```
db : Database = {e1,e2}
```

Figure 13: Creating a database in TDBPL (1).

The (..) operator takes the place of *crtuple* and *addtuple*, constructing a tuple on disk in a single transaction. The {..} operator takes the place of *croset* and *addoset*. If the construction process is re-arranged into a single statement, it is possible for the whole database to be constructed in a single transaction, as depicted in Figure 14.

```
db : Database = {("Smith",23,25000),("Jones",45,18000)}
```

Figure 14: Creating a database in TDBPL (2).

First class closure is now in the control of the TDBPL compiler. The type of each field can be inferred from the schema, and hashing functions are based on declared key fields. The low level characteristics are embedded into the compiler and are no longer the concern of the database programmer.

Using the TDBPL notation, the query defined in Figure 10 can be expressed with increased clarity. This is detailed in Figure 15.

```
Foreach value in db
Return PosInt
Do
  name = value.Name;
  If name = "Smith"
  Then
    Exit With (value.salary)
  Fi
Od
```

Figure 15: Performing a query in TDBPL.

The *Foreach* iterator is an abstraction over the Probe *for_each*. It requires db as the set type, generates a dummy PosInt as the introduction, and the body of the loop is expanded to create the action of the for_each statement. *Exit_for* is equivalent to exiting the iterator with (True, result). *Value.Name* is a re-expression of the *indtuple* operator.

The notation in Figure 15 hides the choice of representation for the above query, where if all the key fields of a tuple are known, but additional attributes are required, then it is possible to directly index the tuple value using a partial representation of the tuple. In the case of Figure 15, it is the decision of the notation compiler to decide whether all the key fields of the tuple are known and generate code using the the *findoset* operator.

4.1.5. Application

The application level is the layer which the database programmer produces. The programmer is presented with a language which supports an object database model that uses statistical analysis to provide the optimized representation for a database, based on analysis of previous operations on the database. In addition, since TDBPL is a super-set of Ten15, it is still possible to integrate the advanced features of Ten15 into the model, such as the storage of first class procedure.

4.1.6. Statistical analysis

Probe provides for statistical analysis of execution of procedures and queries for the top three levels of its architecture. Analysis of the Database level involves the storage of average execution times for distinct types, operations and, in the case of tuples, attributes. A combination of particular type, operation and attribute constitute a *statistics object*. For example, an employee tuple will have a statistics database with the following statistics object, indexed on a combination of the type, operation and field names, i.e.:

```
{ stats_object["employee","create",""]
  stats_object["employee","index","Name"]
  stats_object["employee","index","Age"]
  ...
}
```

In addition, for each statistical object, a list of slowest operations to particular values within the associated data structure is maintained. The list is used as a basis for identifying values and regions which require optimized access.

The statistics module is made available to the language and applications, so that blocks of code, queries and transactions can be associated with statistics objects. At this level, statistics can be used to compare different implementations and algorithms in implementing queries.

4.2. Features of Probe

There are two main design criteria in the construction of Probe: to provide support for a complex data model and to provide efficient management of bulk data. The complex data model is achieved through the expressive power of the Ten15 type system, as discussed in section 3.2. Efficient management of bulk data is managed through a combination of factors, which make the Probe system unique, particularly its provision of *persistent mutation*. Three main branches can be identified: *organisational, representational* and *schema* mutation.

Mutation is a refinement on the ideas of database reorganisation, as found in relational database systems [10], and on storage selection, as found in the object-oriented database system ObjectStore [29], in that the representation of a database changes to match optimum access patterns and that there is control over the representation that a database adopts.

4.2.1. Organisational mutation

Probe supports two kinds of bulk constructor: sets and tuples. A set is effectively an unordered collection of unique values of a particular type. A tuple is a set of name to value mappings, where each name is unique to the data structure. As discussed in section 4.1.6, attributes of tuples and individual values of sets can be identified as having slow read or write times. Organisational mutation requires the identification of values requiring optimization in order to provide a more efficient representation. This form of mutation exploits the dynamic characteristics of a data structure.

Two criteria can be used as a basis for organisational mutation: those values that are frequently accessed thereby overshadowing the importance of access times; and those values that are accessed a 'reasonable' number of times, but access times are extremely long. The vague statements in the previous sentence can be translated into hard thresholds for organisational mutation by the identification of average number of accesses per value and the average access time, and their respective standard deviations, for both reading and writing. An addition factor must take account for the size of the values and the current extent of the bulk constructor, specifically the number of BSAs required to reach the value. This data is brought together into a *mutation lattice*, which is a three dimensional table which maps access times onto value size and the constructor extent.

The actual process of mutation can be invoked after a specific period, after a slow operation on a value or as a separate operation. When a data structure mutates, the statistics from a statistics object are compared against the mutation lattice. Using the value size for the data structure and the current size depth of the representation, the threshold time for mutation will be delivered. If the average access time for the current data structure lies outside the critical region for the threshold, then it is likely that that particular operation on the data structure requires some form of optimization. If the case is the same for all or a majority of permissible operation on the data structure, then an actual alteration to the current representation should be made.

Mutation can take several forms. *Hot spots*, those values that require an optimized access path, can be placed higher in the data structure hierarchy, i.e. at the root, increasing the performance of read and write operations. Also, they can remain in their location but be brought into memory whenever the data structure is first accessed, thereby obviating the need trace the value for a read or a read with intent to write operation. *Active regions* can also be identified, which are group of values or levels in the data structure representation that are frequently accessed or inefficient. Similar optimizations can be applied as for hots spots.

Optimized access brings with it its own problems. Once a value or region is given an optimized access path, it must be possible to determine whether a value still requires it at subsequent invocations of the mutation process. This can be achieved the careful monitoring of performance by assigning a statistics object to the set of optimized values. In addition, there must be a constraint as to the number of optimized access paths that a data structure can support, since the actual process of determining whether the path to a value is through the normal mechanism or is optimized adds an extra overhead to access, and the larger the number of values that are supported, the greater the overhead. This constraint can be user supplied or be determined by the mutation process.

Organisational mutation is being used in sets and tuple. In sets, operations are used as the main analysis criteria and accesses to values that are inefficient are maintained in the slow lists of each operation object. If a value appears in the slow list of all operations, then it is mutated, provided that it meets the mutation threshold as defined in the mutation lattice. Tuples provide a greater control over mutation since inefficient access to keys and operations can be used as a basis for statistical analysis. Hence mutation takes the form of analysising operations for each key, and mutating accordingly if the key meets the threshold and is inefficient for a majority of operations.

4.2.2. Representational mutation

Representational mutation is an attempt to exploit the initial and static characteristics of a bulk data constructor to provide the optimum efficient representation. It is the mutational policy which can be applied to data structures which have an known initial representation, such as vectors and arrays, or have fixed upper bounds, such as bounded sets, bounded buffers.

Vectors and arrays are examples of the atomic data structures that Probe supports. An atom is a data type whose representation is beyond the scope of the data model. Since they may also be constructors for bulk data, and by the orthogonal laws of persistence, have the right to be persistent, it is necessary for them to have a persistent representation which does not inhibit the performance of the data structure. Arrays and vectors can make use of the data structures provided by the Persistence level as a basis for a persistent representation.

The characteristics of arrays and vectors are that they are created to be a fixed size, with each value referring to a constant value. These characteristics can be exploited by representational mutation. Accesses and additions to arrays and vectors can be described in terms of a *life-cycle* model. On creation, a data structure is constant, where every value has the same type and value. As indexes in the data structure are altered, an anomalous stage is reached, where the data structure can be represented in its original format, with a set of anomalous value to index pairs. As more indexes are altered, the data structure becomes sparse, whereby the original representation is no longer valid, since a large number of the indexes have been changed, and it is inefficient to continue searching the value to index pairs. The final stage in the life cycle is when the data structure becomes dense, so that no trend in the values that the structure contains can be identified. The life cycle is reversible, in that it is possible for a data structure to become constant over time, by gradually altering all the changed indexes back to the original value, or indeed some other value.

Every stage in the life cycle can have several representations. The constant phase can be represented as a constant value paired with a set of bounds for indexes, as an alternative to the obvious representation of an allocated contiguous block of constant values. The anomalous phase can make use of the constant representation, pairing it with a set of value to index pairs, where the indexes must be within the range of the bounds defined in the constant form. The sparse form can have several, equally effective representations. One way is to decompose the data structure into a number of regions, making use of the anomalous format to represent each region. Another way is to extend the anomalous format, altering the storage for the index pairs from a value domain to a regional domain. The dense phase is the full representation of the data structure in a format suitable for persistent storage, i.e. making use of the primitive data structures provided by the Persistent level of Probe.

A data structure can be seen as a power set of representations, connected by transformation processes. Using Ten15, it is possible to model the alternative representations as unions, and manipulate them in the abstract data types of the Database level of Probe.

As an example of representational mutation, we consider the case of modelling a persistent vector of values. The data type is represented by a contiguous block of values, indexed by a numeric key, residing in persistent storage. The create operation for the data type takes an upper bound, an initial bound and a single value, returning a block comprising the initial value. The data type is ripe for application of the life cycle model of representational mutation, since it clearly has an initial configuration distinguishable from its intended format. Figure 16 demonstrates in graphic form the alternate representations possible for a persistent vector. During its constant phase, it can make use of the format described above, i.e. pairing the value with the bounds. An anomalous format would be to use the constant representation and to extend it with a set of index to new value pairs. Depending on the size of the vector, the sparse format could be derived from the anomalous format, using regional decomposition of the data structure, or it could be ignored if the data structure is small. In the dense stage, the vector could be represented as either a persistent block, if the data structure is only a few blocks in size, or using the primitives provided by the Persistent level. Indexing and assigning the mutable vector relies on the the operations of the data type understanding the different possible representations. Changing between the different stages in the life cycle of the data structure is governed by statistical analysis of the efficiency of access and update to the formats.

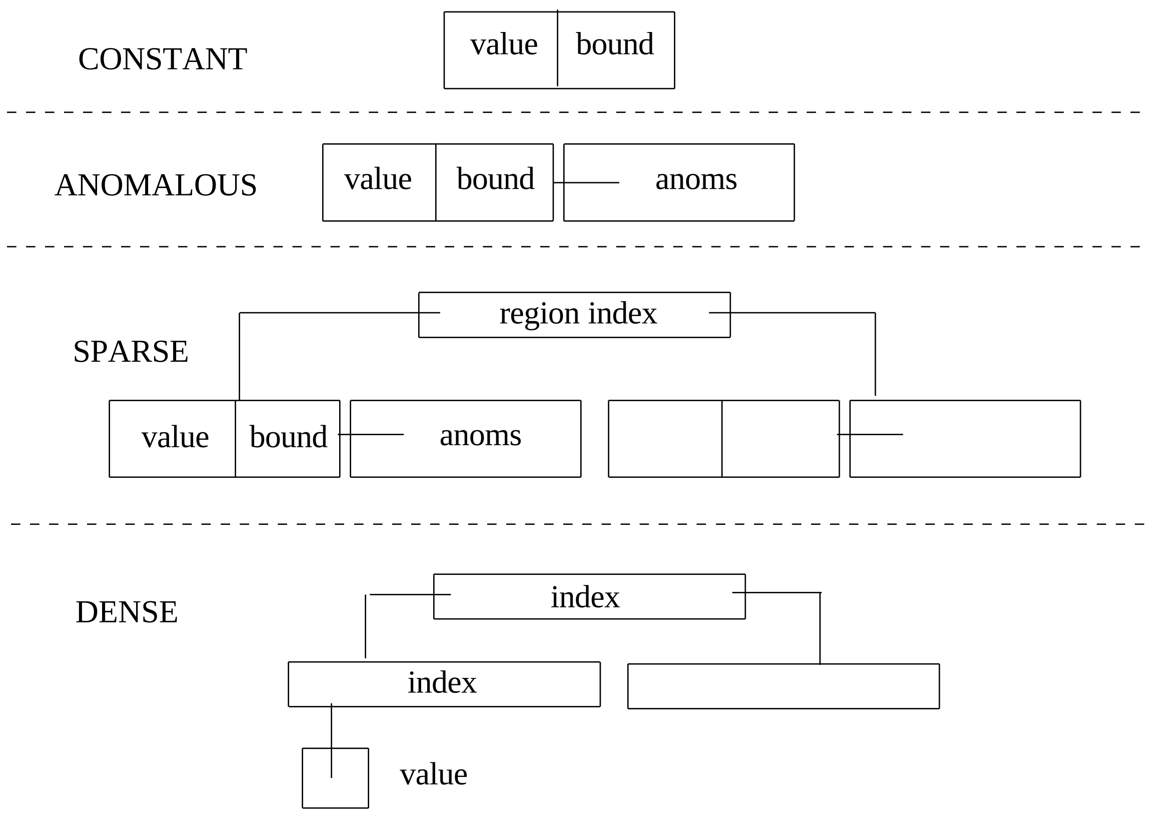

Figure 16: Representational mutation applied to a persistent vector.

Representational mutation can also be applied to data structures that are constructed dynamically but have a fixed set of bounds imposed at initialisation time, i.e. a set which can have most ten values. The policy of mutation in this case is one of restriction, in that it is not necessary to provide a full dynamic representation for the data structure and that an alternative representation can be used. In the case of the set with ten values, instead of using an extensible hash table to represent it, it will be more efficient to use the normal persistence mechanism since the representation is unencumbered by the need to resolve hash indexes or have the overhead of supporting data structures. If the bounded set is larger, then a limited form of decomposition of the indexes can be used to optimize accesses. When the bound become extremely large or the restriction is removed then the extensible hashing based representation can be used. The thresholds for each level are held within the mutation lattice, but a general rule for this form of mutation is that if the maximum size of the data structure is less than the cluster size (a collection of disk blocks) then a flat representation is the most efficient.

Given the hypothesis that persistent mutation is desirable, a problem remains concerning when to transform between the different representations for a data structure. The problem can be solved by analysing the performance of operations on the data structures. In the case of representational mutation, when accessing a particular representation becomes inefficient, the data structure should adopt a different, hopefully more efficient representation. The thresholds for representational mutation can be kept in a mutation lattice format as discussed in the previous section, where optimal access times are mapped against the size of data being stored and variation in the bounds and dimensions of the specific data structure.

Representational mutation can be found in the Database level atomic data structure SVEC (static vector) which is incrementally transformed form a constant representation to a dense version and in the DICTIONARY (dynamic tuple) data structure, where key to value pairs are added dynamically to a dictionary. In addition, it is possible to derive bounded sets from Usets and Osets by simple modification to their sources, allowing for persistent vectors of values if the size of the data structure will not exceed the size of a cluster.

4.2.3. Schema Mutation

A third form of mutation is provided by the Probe system. *Schema mutation* affects the Database data structure available in the Database level of Probe. Generally, the Database data structure collects together all object of the same type into a database. In addition, the Database construct allows for objects with slightly different schemas to be present in the database. This is because database schemas are not constant: as the use and needs of a database change over its lifetime, so does its schema. The Database data structure is implemented as a set of schema to set of object pairs, where the schema is a type representation of the objects contained in the set. Operations on the database are bound to a particular schema by closure of the schema type representation, so that errors resulting from applying operations to schemas that do not match the one expected by the operation cannot occur. The usefulness of this facility is that it allows several different version of a database to co-exist, alleviating the need to change every item in a database whenever a schema modification is made†.

Where schema mutation differs from organisational and representational mutation is that it is under the control of the database programmer to change the schema of a database, and is undertaken at compile time, whereas the other forms are hidden from the programmer, and the changes in representation are carried out at run-time based on statistical analysis of operations on the data structures.

4.3. Satisfaction of requirements

In section two, we listed our requirements for the a system to support an efficient object database model. We now examine the extent to which Probe meets the requirements.

Rich data model – Probe supports an object database model, allowing aggregation and generalisation through sets and tuples, which can be nested to an arbitrary degree of complexity. Indeed the Ten15 type system allows for recursively nested databases to be built. For example, a family tree can be built up as a recursive data model, where an ancestor is a name and a set of descendents, and each descendent is an ancestor to the next

† The inspiration for schema mutation stems from reference [6].

generation:

> **Ancestor = TUPLE [NAME, SET {Descendent}]**
> **Descendent = Ancestor**

This can be expressed using the Probe object database model as:

> **Ancestor = TUPLE**
> **(* has two fields, the second field of existential type Descendent *)**
> **Descendent = SET [Ancestor]**

Persistence – Probe supports orthogonal persistence due to its integration to Ten15. Also, it allows the construction of persistent data structures that can be accessed efficiently, and are controlled with respect to reading and writing.

Concurrency – The support that Probe has for concurrency depends on the model provided by Ten15. Ten15 has a well developed concept of transactions on persistent variables. This can be incorporated into the Probe model by extending the notion of writeable data structures to include an active transaction. This means that whenever a data structure is to be rewritten, the operation should either create a new transaction or be added to an existing transaction. Subsequent operations on associated data structures can then be added to the transaction. Finally, when the transaction is complete, the transaction can either be committed or aborted, without fear of the integrity of the database being compromised. Transaction-based write operations can exist in parallel with operations which assign and commit a persistent variable in a single action.

Placement of persistent variables on distributed datastores can also be managed by Probe using the facilities provided in Ten15. Ten15 provides anonymous references to values on the persistent store, so these can easily be made to refer to local or remote persistent data. In addition, each datastore has an associated write capability. Persisting a value with a specific capability defines the placement of the value in the distributed datastore. If statistical analysis within Probe determines that it would be beneficial for the performance of a particular process to have a remote persistent data structure stored on a local datastore, by using the write capability to the local datastore, the data structure could be rewritten onto the local store, thereby providing object migration and improved access performance.

Transactions are integrated into Probe by the provision of a shadow set of operations which require the provision of an active transaction. If the persistent object is in the transaction, then operations affect the intended reassignment (i.e. a persistent value) to the object which is stored within the transaction, otherwise the pvar is added to the transaction, if it is not currently held by another transaction. The Ten15 primitive *commit_transaction* assigns all persistent values to their respective pvars, updating the root of the datastore in a single operation.

Bulk data – The persistent data constructors of the data structure level permit the efficient management of bulk data due to a principle of decomposing data structures into nested groups of data that fit into one disk block or clusters of disk blocks.

Extensibility – As mentioned above, the modular approach that was adopted to develop Probe means that the system can be extended by making use of the abstract structures provided at every level, and changes can be made to the system with no noticeable changes of other parts of the system.

In the introduction, four reasons for the unsuitability of persistent programming languages to database programming were given. To some extent, Probe eliminates these problems. First, the problem that programming language data structures fails to capture the semantics of a particular data structure is overcome by providing a complex description of a database data structure which allows the data structure to be viewed in a number of ways, depending on its current use and access characteristics.

Secondly, the complex description of a data type permits bulk data management. This is due to the expressive power of the Ten15 type system, since it is possible to represent the necessary decompositions of the data type onto the persistent store at a type system level. In addition, mutation and changes in representation can be used to simulate and optimize the storage of bulk data.

Thirdly, concurrency is provided by the transaction and process primitives available in Ten15. The process primitives allows several processes to exist in parallel on a single node. The transaction primitives provide a mechanism for concurrent process to synchronise their access and update of the Ten15 datastore in an integrity preserving manner. The primitives, in no small way, contribute to the possibility of changing the style in which persistent programming languages are programmed. By building up transactions incrementally and allowing for a single commit to access all persistent variables at a stroke, the problems of simultaneous update of a persistent variable and of having partially inconsistent databases due to a single value update, are avoided. The support that Ten15 gives for transactions also alleviates the fourth problem, concerning the differing programming styles of DBPLs and PPLs. Procedures can be used as the basis for concurrent processes and the transaction mechanism can be used to incrementally build transactions from separate procedures, guaranteeing datastore integrity thanks to a single commit operation.

4.4. Current status

Probe is still in its development stage. An initial prototype currently exists which implements the full object database model. Experimentation with the prototype, and statistical analysis, will enable us to evaluate the costs and the benefits of mutation.

TDBPL is a conceptual database programming language, since Probe does not require a notation in order for it to be programmed, and so the development of TDBPL is beyond the scope of this project.

Ten15 is still in its development stages. Concurrent access to shared data is provided for, but concurrent processes are still in development stages. Hence Probe is being implemented as a single process, multi-transaction system.

5. Conclusion

This report has provided a detailed overview of on-going research into the Probe project. It has suggested that the paradigms of databases and persistent programming languages can be integrated in order to provide an extensible environment for the production of database applications. Probe is an example of such an integration of the two paradigms. It exhibits the functionality of the object database model and the flexibility of the Ten15 programming notation. In addition, it has mechanisms for the efficient storage of bulk data, dynamic changes to schemas and allows the incorporation of new data structures, mechanisms and algorithms. The mutation mechanism provides, so far as we are aware, a unique capability for dealing with evolving, complex data structures. Also, the Probe abstract machine provides an ideal basis for highly expressive database programming languages such as TDBPL.

Probe is an effective attempt at unifying the database and persistent programming language paradigms. This is due, in no small way, to the expressive power of the Ten15 type system, the simplicity of its persistence model and the ability to optimize access paths to data using persistent mutation.

6. Acknowledgements

The authors wish to thank Simon Dobson and Derek Bridge for their help and comments during the production of this report.

7. References

1. M. Adabi and L. Cardelli, ‘‘Dynamic Typing in a Statically Typed Language’’, 47, Digital Systems Research Center (June 1989).
2. R. Agrawal and N.H. Gehani, ‘‘ODE (Object Oriented Database and Environment) : The Language and Data Model’’, pp. 36-45 in *ACM Sigmod Proceedings of the International Conference on the Management of Data* (1989).
3. A. Albano, L. Cardelli and R. Orsini, ‘‘Galileo : a Strongly Typed Interactive Conceptual Language’’, *ACM Transactions on Database Systems* **10**(2), pp. 230-260, ACM (March 1985).
4. A. Albano, A. Dearle, G. Ghelli, C. Marlin, R. Morrison, R. Orsini and D. Stemple, ‘‘A Framework for Comparing Type Systems for Database Programming Languages’’, pp. 170-178 in *Second International Workshop on Database Programming Languages*, Morgan Kaufmann (1989).
5. M.C. Atkins, ‘‘Implementation Techniques for Object-Oriented Systems’’, YCST 90/01, Dept. of Computer Science, University of York (June 1989). DPhil Thesis.
6. M. Atkinson, ‘‘Questioning persistent types’’, pp. 2-24 in *Second International Workshop on Database Programming Languages*, Morgan Kaufmann (1989).
7. M. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott and R. Morrison, ‘‘An Approach to Persistent Programming’’, *B.C.S. Computer Journal* **26**(4), pp. 360-365 (November 1983).
8. M. Atkinson and O.P. Buneman, ‘‘Types and Persistence in Database Programming Languages’’, *ACM Computing Surveys* **19**(2), pp. 105-190 (June 1987).
9. M. Atkinson, P. Buneman and R. Morrison, ‘‘Binding and Type Checking in Database Programming Languages’’, *B.C.S. Computer Journal* **21**(2), pp. 99-109 (April 1988).
10. D.S. Batory, ‘‘Optimal File Design and Reorganisation Points’’, *ACM Transactions on Database Systems* **7**(1), pp. 60-82 (March 1982).
11. M. Brandreth, P.W. Core, I.F. Currie, N.E. Peeling, M. Stanley and J.M. Foster, ‘‘Ten15 Prototype’’, R.S.R.E. Report 91025 (1991).
12. A.L. Brown, ‘‘Persistent Object Stores’’, Persistent Programming Research Report 71, Universities of Glasgow and St. Andrews Computer Science Departments (March 1989).
13. P.A. Buhr, G. Ditchfield and C.R. Zarnke, ‘‘Basic Abstractions for a Database Programming Language’’, pp. 422-438 in *Second International Workshop on Database*

Programming Languages, Morgan Kaufmann (1989).

14. L. Cardelli, ‘‘A Polymorphic Lambda Calculus with Type : Type’’, 10, Digital Systems Research Center (May 1986).

15. L. Cardelli and P. Wegner, ‘‘On Understanding Types, Data Abstraction and Polymorphism’’, *ACM Computing Surveys* **17**(4), pp. 471-522 (December 1985).

16. W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey and R. Morrison, ‘‘Persistent Object Management System’’, *Software - Practice and Experience* **14**(1), pp. 49-71 (January 1984).

17. E.F. Codd, ‘‘A Relational Model of Data for Large Shared Data Banks’’, *Communications of the ACM* **13**(6), pp. 377- (June 1970).

18. D. Comer, ‘‘The Ubiquitous B-Tree’’, *ACM Computing Surveys* **11**(2), pp. 121-137 (June 1979).

19. G. Copeland and D. Maier, ‘‘Making Smalltalk a Database System’’, pp. 316-325 in *ACM Sigmod Proceedings of the International Conference on Management of Data* (August 1984).

20. A. Dearle, ‘‘Environments: A flexible binding mechanism to support system evolution’’, Persistent Programming Research Report 67 , Universities of Glasgow and St. Andrews Computer Science Departments (1988).

21. A. Dearle, ‘‘A Persistent Architecture Intermediate Language’’, Persistent Programming Research Report 35, Universities of Glasgow and St. Andrews Computer Science Departments (1987).

22. K. Dittrich, *Damokles Reference Manual*, University of Karlsruhe, Karlsruhe, W. Germany (1989).

23. R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong, ‘‘Extensible Hashing - A Fast Access Method for Dynamic Files’’, *ACM Transactions on Database Systems* **4**(3), pp. 315-344 (September 1979).

24. J. M. Foster, ‘‘The Algebraic Specification of a Target Machine: Ten15’’, in *High Integrity Software*, ed. C.T. Sennett, Pitman (1988).

25. K.H. Goodenough and S.J. Rees, *A Notation for Ten15*, Royal Signals and Radar Establishment, Malvern (May 1989).

26. J.M. Smith and D.C.P. Smith, ‘‘Database Abstractions: Aggregation and Generalization’’, *Communications of the ACM*, pp. 138-151 (January 1977).

27. E. Jul, H. Levy, N. Hutchinson and A. Black, ‘‘Fine-grained mobility in the Emerald system’’, *ACM Transactions on Computer Systems* **6**(1), pp. 109-133 (February 1988).

28. S. Khoshafian, ‘‘A persistent complex object database language’’, *Data and Knowledge Engineering* **3**(4), Ashton Tate, Walnut Kreek, CA, USA (February 1989).

29. C. Lamb, G. Landis, J. Orenstein and D. Weinreb, ‘‘The ObjectStore Database System’’, *Communications of the ACM* **34** (10) (October 1991).

30. F. Mathas and J.W. Schmidt, ‘‘The Type System of DBPL ’’, pp. 219-225 in *Second International Workshop on Database Programming Languages*, Morgan Kaufmann (1989).

31. R. Morrison, F. Brown, R. Connor and A. Dearle, ‘‘The Napier88 Reference Manual’’, Persistent Programming Research Report 77, Universities of Glasgow and St. Andrews

Computer Science Departments (July 1989).

32. N. E. Peeling and K. R. Milner, ‘‘KeepSake: A Database Kernel’’, R.S.R.E. Memorandum 88014 (March 1989).

33. J.E. Richardson and M.J. Carey, ‘‘Persistence in the E language : Issues and Implementations’’, *Software - Practice and Experience* **19**(12), pp. 1115-1150 (December 1989).

34. J. W. Schmidt, ‘‘Some High Level Language Constructs for Data of Type Relation’’, *ACM Transactions on Database Systems* **2**(3), pp. 247-261 (1977).

35. David W. Shipman, ‘‘The Functional Data Model and the Data Language DAPLEX’’, *ACM Transactions on Database Systems* **6**(1), pp. 140-173, ACM (March 1981).

36. A. Straw, F. Mellender and S. Riegel, ‘‘Object Management in a Persistent Smalltalk System’’, *Software - Practice and Experience* **19**(8), pp. 719-738 (August 1989).