Existentially Quantified Types as a Database Viewing Mechanism

Richard Connor, Alan Dearle, Ronald Morrison & Fred Brown

richard%uk.ac.st-and.cs@ukc
al%uk.ac.st-and.cs@ukc
ron%uk.ac.st-and.cs@ukc
ab%uk.ac.st-and.cs@ukc

Department of Computational Science
University of St Andrews
North Haugh
St Andrews
Scotland
KY16 9SS

Abstract

In database management systems, viewing mechanisms have been used to provide conceptual support for the user and security for the overall system. By controlling the style of database use, views aid the user by concentrating on the area of interest, and the system by ensuring the integrity of the data.

In recent years, there have been a number of proposals for database or persistent programming languages. Such languages must be able to provide the facilities traditionally found in both database management systems and programming languages. In this paper we demonstrate how a persistent programming language, Napier88, can provide a viewing mechanism over persistent data by using existentially quantified types. The views, which may be generated dynamically, are statically type checked. The technique shown is applicable to any language which supports existentially quantified data types and a persistent store.

1 Introduction

Viewing mechanisms have been traditionally used in database systems both to provide security and as a conceptual support mechanism for the user. Views are an abstraction mechanism that allow the user to concentrate on a subset of types and values within the database schema whilst ignoring the details of the rest of the database. By concentrating the user on the view of current interest both security and conceptual support are achieved.

Persistent programming languages [ACO85,Mat85,ps87] integrate both the technology and methodology of programming languages and database management systems. One particular area of difficulty in this integration has been the friction caused by the type mechanisms of programming languages and databases being incompatible. Programming languages tend to provide strong, often static type systems with little to aid the expression or modelling of large uniform structure. Databases, on the other hand, are much more concerned with capturing this notion of bulk expression than with static or even strong typing. Both, however, are concerned with the integrity of the data.

Another area of difficulty in the integration of programming languages with database systems has been to demonstrate that the facilities found necessary in both systems are not lost or compromised by the integration. Viewing mechanisms which have been used so successfully in the database community have not found widespread acceptance in the programming language world. They therefore constitute a source of irritation in the above integration.

It is the intention of this paper to demonstrate how the facilities of the persistent programming language, Napier88 [MBC88], may be used to provide a viewing mechanism. Napier88 provides a strong polymorphic type system with existentially quantified types and a persistent store. In particular, it will be shown how existentially quantified types may be used to provide multiple views over objects whilst retaining static type checking.

2 Viewing mechanisms

Viewing mechanisms are traditionally used to provide security and information hiding. Indeed, in some relational database systems, such as INGRES [SWK76], a relational viewing mechanism is the only security mechanism available. A view relation is one which is defined over others to provide a subset of the database, usually at the same level of abstraction. A slightly higher level may be achieved by allowing relations to include derived data, for example, an age field in a view might abstract over a date of birth field in the schema.

Security provided by view relations is often restricted to simple information hiding by means of projection and selection. For example, if a clerk is not permitted to access a date of birth field, then the projected data may contain all the attributes with the exception of this one. If the clerk may not access any data about people under the age of twenty-one, then the view relation will be that formed by the appropriate selection.

Read-only security may be obtained in some database systems by restricting updates on view relations. Although this restriction is normally due to conceptual and implementation problems, rather than being a deliberate feature of a database system design, it may be used to some effect for this purpose. Some systems, for example IMS [IBM78], go further than this, and the database programmer can allow or disallow insertions, deletions, or modifications of data in view relations. This allows a fine grain of control for data protection purposes.

There are a number of problems associated with view relations. Often a new relation is created, with copied data. This means that updates (where permitted) to the view relation will not be reflected in the underlying relations, with the obvious loss of integrity. This problem may be partly solved by the use of delayed evaluation, where names are not evaluated until used in a query. This allows the database programmer to define views over the database schema before all of the data is in place.

Those systems where data copying does not occur seem to incur major integrity problems. For this reason, System R [ABC76] disallows updates on views which are constructed using a join operation. There is a more serious example in IMS, where deletion of a segment, if permitted, also deletes all descendant segments, including cases where they are not even a part of the view in question!

A much higher-level concept of a viewing mechanism is provided by the UMIST Abstract Data Store [Pow85]. This is a software tool which supports abstract data objects together with mechanisms for providing different views over them. The system provides a consistent graphical user interface which allows the user to directly manipulate objects via multiple views. Changes in objects are automatically reflected in other views since they contain no information about the state of objects, only the manner in which they are displayed. This provides a powerful tool for a user to build a store of objects which may be maintained and incremented interactively, and it seems possible that it may be sufficient for a surprisingly large class of problem. However, the inability to write general-purpose programs over the store must be seen as a major drawback for many applications.

3 Existentially quantified types in Napier88

Napier88 provides the existentially quantified types first described by Mitchell & Plotkin [MP85]. These types are often called abstract data types since the objects of such a type display some abstract behaviour that is independent of the representation type.

There are two important properties of the Napier88 abstract data types. The first is that users should not be allowed to break the abstraction via the abstract data type interface. Thus, once an abstract object is made, its representation type can never be rediscovered; it is truly abstract. The second property of these types is that they are statically type checked.

Abstract data types in Napier88 will be introduced by example. Care will be taken to explain the elements of the abstract data type in order that the reader is not lost in the syntax of Napier88.

The type declaration,

type TEST **is abstype**[i](value : i ; change : **proc**(i → i))

declares the identifier *TEST* as a type that is abstract. The identifier enclosed in square brackets is called the witness type and is the type that is abstracted over. There may be one or more witness types in any abstract data type. In this case, the abstract data type interface consists of an identifier *value* with the type *i* and an identifier *change* which is a procedure that takes a parameter of type *i* and returns a result of type *i.* This type is written **proc**(*i* → *i*).

To create an abstract data type the fields in the interface are initialised. For the above type *TEST* an object of type *i* and another of type **proc**(*i* → *i*), for some *i*, are required. There follows an example using the type integer as the representation type. Firstly, an increment procedure for integers may be written as,

let incInt = **proc**(x : **int** → **int**) ; x+1

The reserved word **let** declares an object, with the identifier *incInt* to be a procedure that takes an integer and returns it incremented by 1.

Having created the *incInt* procedure it may be used as a *change* procedure in an instance of *TEST* by the following declaration,

let this = TEST[**int**](3,incInt)

which declares the object *this* to have the abstract type *TEST*, the concrete witness type **int**, the *value* field initialised to 3 and the *change* field initialised to the procedure *incInt.* Once the abstract data object is created, the user can never tell that the representation type is **int**. The object can only be used through its abstract interface.

The declaration,

let that = TEST[**int**](-42,incInt)

creates another abstract data object of the same type. However even although *this* and *that* have the same representation type the user may never discover this fact and cannot make use of it.

Finally,

let incReal = **proc**(x : **real** → **real**) ; x + 1.0

declares an increment procedure for real numbers, and,

```
let also = TEST[ real ]( 42.0,incReal )
```

declares another abstract data object with the same type, namely *TEST*, as *this* and *that* even although it has a different representation type.

These abstract data types display second order information hiding [CW85]. In first order information hiding a hidden object is encapsulated by scope and manipulated by procedures that do not mention the hidden object as parameters or results. In second order information hiding, the type of the hidden object is defined along with operations on it in an abstract interface. Thus it can be referred to in the abstract and instantiated with a concrete type. We will demonstrate some of the advantages of second order hiding later.

It should be mentioned here that Napier88 uses structural type equivalence semantics and therefore all abstract types with the same structure, and not just type name, are type equivalent. This property will be used in type checking across the persistent store for independently prepared program and data [ABM88].

As we have seen above, the representation type of different instances of an abstract data type may be different. Therefore the implementation must ensure that operations from one object are never applied to another. Also since, in general, the compiler cannot determine the representation type statically, the abstract data object may only apply its operations to itself. This is sufficient for static type checking.

To manipulate an abstract data object, the **use** clause is utilised. For example,

```
use this as X in
begin
      X( value ) :=  X( change )( X ( value ) )
end
```

defines a constant identifier, in this case *X*, for the abstract object, in this case *this*. This binding is necessary for two reasons. Firstly, the object can be expressed as any legal expression and may therefore be anonymous. Secondly, if the object is not anonymous, an assignment may be made to it within the scope of the **use** clause which could invalidate the static type checking. The body of the **use** clause alters the *value* field of the abstract data object by applying the *change* procedure with the *value* field as a parameter and assigning the result back to the *value* field.

A procedure to abstract over the abstract data type may be defined as follows:

```
let changeTEST = proc( A : TEST )
      use A as X in
            X( value ) : = X( change )( X ( value ) )
```

with calls,

```
changeTEST( this )
changeTEST( that )
changeTEST( also )
```

The *changeTest* procedure may be legally applied to any data object that is structurally equivalent to *TEST*, and changes the *value* field of the abstract data type by applying the *change* procedure to it.

This concludes the introduction to abstract data types except to mention that it is often useful to name the abstract witness types in a **use** clause. This can be achieved by placing their identifiers in square brackets after the constant binding identifier. However, no use will be made of this in this paper.

The essence of our viewing mechanism is to take a body of data, a database, place it in an abstract data type and give it to a user to manipulate in this abstract form. Since the same data or its subsets may be placed in many abstract data types it is possible to have many possible types or views on the same data. Most importantly the views can be statically type checked. We will give a detailed example of this later. For the present we will describe the environment mechanism of Napier88 for provision of a persistent store and a convenient mechanism for holding the data.

4 Environments in Napier88

In Napier88, environments are collections of bindings which may be extended or contracted under user control [Dea89]. Each binding contains an identifier, a value, a type and a constancy indicator. By manipulating the bindings in environments the user can control the name space of the data in the bindings. In particular, environments are used by convention in the persistent store as repositories for program and data that may be incrementally and independently generated. Thus, they are used in the construction and composition of systems out of components in the persistent store. Furthermore, the manner in which they are used controls the name space in the persistent store.

All environments belong to the same infinite union type, denoted by the typename **env**. To use the bindings in an environment a projection from the infinite union must take place.

Again environments will be introduced by example. To create an environment the standard procedure *environment* is called. It has the type:

proc($\rightarrow$ **env**)

Calling this procedure creates an environment containing no bindings. Adding a binding to an environment is the dynamic analogy of adding a declaration (binding) to a scope level. Indeed, it will be shown later how environments may be used to dynamically compose scope levels. Bindings are therefore added to an environment by declarations. For example,

```
let firstEnv = environment( )
in firstEnv let absInc = this
```

will create an environment, called *firstEnv*, and place the binding *absInc* with value *this*, which is the abstract data object defined above, in it. The environment records for each binding: the identifier, the value, the type and a constancy indicator. In this example, the environment will record: the identifier *absInc*, the value *this*, the type *TEST* and the constancy indicator **true**. The binding is added to the environment but not to the local scope.

Environments may be extended by further declarations and may be contracted by a **drop** clause which is not used here.

To use the bindings in an environment, a projection statement called a **use** clause is invoked. The environment **use** clause, which is different from the **use** clause for abstract data types, projects a binding into the local scope. For example,

```
use firstEnv with absInc : TEST in <clause>
```

binds the identifier *absInc* into scope as a declaration at the head of <clause>. The value of *absInc* is the value in the environment. Alterations to the value will alter the value in the environment.

The environment **use** clause need only specify the particular bindings that are to be used. Others that are present but not specifically mentioned are not in scope and therefore not usable within the <clause>.

As mentioned earlier, environments are used to impose structure on the persistent store. In accordance with the concept of orthogonal persistence [ABC83], all data objects in Napier88 may persist. For each incarnation of the Napier88 object store there is a root of persistence which may be obtained by calling the predefined procedure *PS* which has type:

```
proc( → env )
```

Thus, the distinguished root of the persistent store graph is of type **env**. When a program is activated, the distinguished root will contain all the bindings for that universe. The standard bindings are defined for every system and one of these bindings contains the procedure used earlier to create environments. It is bound to the identifier *environment*. The following program illustrates the use of this procedure:

```
let ps = PS()
use ps with environment : proc ( → env ) in
begin
        let newEnv = environment()
        let this = TEST[ int ]( 3,proc( x : int → int ) ; x + 1 )
        in NewEnv let absInc = this
end
```

This program binds the root of persistence to the local identifier *ps*. The first **use** clause binds the identifier *environment* to the environment creation procedure in the root environment. Inside the body of the **use** clause this procedure is called to create a new (empty) environment. Finally a binding to *this*, denoted by *absInc*, is made in the newly created environment. The reader should note that the environment bound to *newEnv* is not yet persistent. Objects that persist beyond the activation of the unit that created them are those which the user has arranged to be reachable from the root of persistence. To determine this, the system computes the transitive closure of all objects starting with the root. Thus, in order to store objects in the persistent store the user has to alter or add bindings that can be reached from the distinguished root. In order to make the environment *newEnv* persist the above example may be rewritten as:

```
let ps = PS()
use ps with environment : proc ( → env ) in
begin
        let localNewEnv = environment()       ! Create an environment, bound to localNewEnv
        let this = TEST[ int ]( 3,proc( x : int → int ) ; x + 1 )
        in localNewEnv let absInc = this       ! in the local scope. Place 'this' in localNewEnv.
        in ps let newEnv = localNewEnv         ! Finally, bind localNewEnv to newEnv in ps.
end
```

At the end of this program the abstract data object *absInc* will be bound to the environment *newEnv* which is in the root environment.

The object may be retrieved from the persistent store and used by the following code:

```
use PS() with newEnv : env in
use newEnv with this : abstype[ i ]( value : i ; change : proc( i → i ) ) in
        use this as X in
        X( value ) := X( change )( X ( value ) )
```

Both abstract data types and environments will now be used to construct a viewing mechanism. The strategy is to place the raw data of a database in an environment or environments where it may persist. Within these environments, the data will be placed in an abstract data type that is appropriate to a particular view and that abstract data type stored in the persistent store. The views, or abstract data type interfaces may be exported to appropriate users. Since the raw data may be placed in many abstract data types, multiple views may be constructed and retained.

Higher order views may also be constructed by using existing views as components of the new abstract data types. These views may be used and made to persist in the usual way. Furthermore, database views often include a notion of looking at a subset of tuples or objects in a collection, in addition to limiting the protocol to individual objects. As the views described here are written in a general purpose programming language, this may be achieved by limiting the range of the procedure which retrieves the data from the database so that it does not return data which is outside the view in question.

5 Example of view construction

For a programming example, we will use a banking system in which customers have access to their accounts through autoteller machines. This poses the classic database problems of having a very large bulk of updatable data with different users requiring different operations on it. Here we will concentrate on views in the system. We shall restrict ourselves at first to the autoteller machines, which have different styles of access to accounts according to which bank the machine belongs. A customers own bank may have full access to an account whereas another bank may not access the customer's account balance, but must know if a withdrawal may be made.

5.1 Creating a concrete representation

We will begin by defining a concrete representation of a user account and the procedures that operate on that type. Here we are not interested in how the account data is stored (i.e. in a relation, a β-tree, etc.), and we will assume that we have previously declared a lookup function indexed by account number which returns the account associated with that account number.

We will implement an account using a record-like data structure which in Napier88 is called a structure. The structure type, called *account*, has three fields: *balance* which holds the account balance; *limit* which contains the overdraft limit and *pin* which contains the password necessary to access the account. There is one instance of this type in the database for each user account.

One special instance of this type called *failAc* is created to use as a fail value. This is used in the *getAc* procedure if an illegal password is supplied when attempting to access an account.

Five procedures operate on the account directly, they are:

withdraw:	This procedure checks to see that sufficient funds are in the specified account to make a withdrawal. If there are, the amount specified is debited from the account.
balance:	This returns the balance of an account.
limit:	This returns the overdraft limit of an account, this is a negative number.
sufficient:	This indicates whether sufficient funds are in an account to make a withdrawal of a specified amount.
getAc:	This procedure interfaces with the persistent store, it looks up an account in an associative storage structure and checks the supplied password. If the password matches the *pin* field in the account, the account is returned, otherwise the fail value *failAc* is returned.

The Napier code necessary to implement the concrete type representations and the code to operate on the concrete type representation of an account is shown below:

```
type account is structure( balance,limit : int ; pin : string )

let failAc = account( 0,0,"" )                          ! This is a fail value

let withdraw = proc( debit : int ; ac : account )        ! Withdraw debit pounds from ac.
begin
        let result = ac( balance ) - debit
        if result > ac( limit ) and debit > 0 do ac( balance ) := result
end

let balance = proc( ac : account → int ) ; ac( balance )   ! Return the balance of account ac.

let limit = proc( ac : account → int ) ; ac( limit )       ! Return the credit limit of account ac.

let sufficient = proc( debit : int ; ac : account → bool )  ! Return whether or not debit pounds
        ac( balance ) - debit > ac( limit )                  ! may be withdrawn from account ac.

let getAc = proc(  accountNumber : int ;                 ! Look up the account number,
                   passwd : string → account )           ! check user password and if the
begin                                                    ! password matches the password in
        let new = lookup( accountNumber )                ! the database return the account,
        if new( pin ) = passwd then new else failAc      ! otherwise return a fail value.
end
```

As mentioned earlier, we have made use of a predefined *lookup* procedure, as its implementation is of no interest in this context.

5.2 Placing concrete representations in the store

Of course, in a real bank there would be a requirement for the procedures and data defined above to persist over a long period of time. In order to achieve this in practice it is necessary to place them in the persistent store. We will assume that an environment called *bank* already exists in the root environment. The above example may be trivially rewritten as follows:

```
use PS() with bank : env in
begin
      let failAc = account( 0,0,"" )                          ! Must be declared both in local
      in bank let failAc = failAc         ! scope and in bank environment

      in bank let withdraw = proc( debit : int ; ac : account ) ; ...    ! procedure body as above
      in bank let balance = proc( ac : account → int ) ; ...             ! procedure body as above
      in bank let limit = proc( ac : account → int ) ; ...               ! procedure body as above
      in bank let sufficient = proc(   debit : int ;
                                        ac : account → bool ) ; ...     ! procedure body as above
      in bank let getAc = proc(    accountNumber : int ;                ! procedure body as above
                                   passwd : string → account ) ; ...
end
```

Since the environment called *bank* is reachable from the persistent root the procedures will be saved when the program terminates. However, notice that if we allow programmers access to the concrete representations, the database will be vulnerable to misuse. For example, the unscrupulous programmer could write,

```
let myAc = getAc( 34589001,"3478" )
myAc( balance ) := myAc( balance ) + 1000000
```

yielding a net profit of one million pounds (a very high programmer productivity ratio of £500000/line). For this reason, it is necessary to protect the concrete representation with abstract interfaces. In the next section we will show how to do this.

5.3 Creating abstract view types

For the purpose of this example we will define two abstract data types which provide interfaces for the procedures shown in the last section. The first is to be used by an account holders own bank. The abstract type shown below, called *localTeller* has the following five fields:

failAc:	is returned by *getAc* if a password check fails,
getAc:	which will take as input an account number, and a secret password typed into the machine by the customer, and provided that the secret number is correct, return the representation of that account, otherwise it will return the fail value *failAc*,
withdraw:	which will remove the amount specified from the account, unless there are insufficient funds, in which case it will do nothing,
balance :	which returns the balance in the account, and,
limit :	which returns the account overdraft limit.

The type of *localTeller* is defined below:

```
type amount is int          ! an amount of money
type number is int          ! an account number
type passwd is int          ! a secret number

type localTeller is abstype[ ac ]( failAc     : ac
                                   getAc      : proc( number,passwd → ac )
                                   withdraw   : proc( amount,ac )
                                   balance    : proc( ac → amount )
                                   limit      : proc( ac → amount ) )
```

The concrete type named as *ac*, may never be discovered, therefore the programmer is forced to access accounts only using the procedures provided in the interface of the abstract data type. Notice that the fail value *failAc* appears in the interface of the abstract type. This permits the user of an instance of this abstract type to check for failure in the application of the *getAc* procedure. This is only possible because in Napier88 equality is defined over all types, including witness types. Note that it is not possible to write such an abstract type in standard ML [Mil83] since equality is defined only over the so called 'eq' types – a subset of the type domain. In Napier88, equality is always defined as identity

Similarly, we can define another abstract type for use by another bank. Other banks are not allowed to discover a customers' balance or limit so a slightly different interface is required. In this type a procedure called *sufficient* is provided so that the bank may ensure that sufficient funds are available in the account. We may define the type *remoteTeller* as follows:

```
type remoteTeller is abstype[ ac ]( failAc      : ac
                                    getAc       : proc( number,passwd → ac )
                                    withdraw    : proc( amount,ac )
                                    sufficient  : proc( amount,ac → bool ) )
```

5.4 Creating instances of views

Here, an instance of the type *localTeller* and an instance of the type *remoteTeller* are required to act as views over the single implementation shown in Section 5.2. This may be achieved by creating instances of the two types as follows:

```
use PS() with tellerEnv,bank : env in               ! We assume these environments have been
use bank with  failAc : account ;                   ! properly constructed and initialised
               withdraw : proc( int,account ) ;
               balance,
               limit  : proc( account → int ) ;
               sufficient : proc( int,account → bool ) ;
               getAc : proc( int,string → account ) in
begin
      in tellerEnv let local = localTeller[ account ](  failAc,
                                                        getAc,
                                                        withdraw,
                                                        balance,
                                                        limit )

      in tellerEnv let remote = remoteTeller[ account ](   failAc,
                                                           getAc,
                                                           withdraw,
                                                           sufficient  )
end
```

The program extracts the procedures placed in the environment denoted by *bank* and creates an instance of the types *localTeller* and *remoteTeller*. These are initialised using the procedures from the *bank* environment. Therefore, using this technique, the level of procedural indirection normally found in viewing mechanisms is not required. Consequently there are both space and time advantages of this technique.

When this program terminates the two abstract objects denoted by *local* and *remote* in the environment denoted by *tellerEnv* will be committed to the persistent store, since they are within the transitive closure of the persistent root.

Although we have chosen to create the two abstract types used in this example in the same code segment this was not strictly necessary. Any programmer with access to the bank environment is free to create new abstract types which interface with the concrete types at any time in the future in a similar manner.

5.5 Using views

Two views of the database now exist in the environment called *tellerEnv* reachable from the root of persistence. In order to use the instance of *localTeller* in the *tellerEnv* the auto-teller programmer has only to write a main loop which uses the procedural interface correctly. This would look something like:

```
use PS() with tellerEnv : env in
use tellerEnv with local : localTeller in

use local as package in
begin
        let getAccount = package( getAc )
        let withdraw = package( withdraw )
        let findBalance = package( balance )
        let findLimit = package( limit )

        ! code to use above procedures ...
end
```

Similarly, the instance of *remoteTeller* could be accessed as follows:

```
use PS() with tellerEnv : env in
use tellerEnv with remote : remoteTeller in

use remote as package in

begin
        let getAccount = package( getAc )
        let sufficient = package( sufficient )
        let withdraw = package( withdraw )


        ...
end
```

The interesting point here is that these procedures manipulate the same objects as those in the previous interface, and so provide a different view of them. If the account information is kept in a relational data structure, this is equivalent to a relational viewing mechanism: *getAc* is a relational select on a primary key. Notice that although the views update the same data, no integrity problems arise.

5.6 N-ary procedures

Suppose another function had been required from the auto-teller, so that a customer with two accounts may use the machine to transfer money from one to another. We can allow this by redefining the interface of *localTeller* as follows:

```
type localTeller is abstype[ ac ]( failAc   : ac
                                   getAc    : proc( number,passwd → ac )
                                   withdraw : proc( amount,ac )
                                   transfer : proc( amount,ac,ac )
                                   balance  : proc( ac → amount )
                                   limit    : proc( ac → amount ) )
```

and by writing the *transfer* procedure in the module which places the interface procedures in the store as follows:

```
use PS() with bank : env in
begin

        let failAc = account( 0,0,"" )              ! Must be declared both in local
        in bank let failAc = failAc                 ! scope and in bank environment
        in bank let withdraw = proc( debit : int ; ac : account )
        begin
                let result = ac( balance ) - debit
                if result > ac( limit ) and debit > 0 do ac( balance ) := result
        end

        in bank let balance = proc( ac : account → int ) ; ac( balance )

        in bank let limit = proc( ac : account → int ) ; ac( limit )

        in bank let sufficient = proc( debit : int ; ac : account → bool )
                ac( balance ) - debit > ac( limit )

        in bank let transfer =proc( amount : int ; from,to : account )
        if amount > 0 and from( balance ) - amount > from( limit ) do
        begin
                from( balance ) := from( balance ) - amount
                to( balance ) := to( balance ) + amount
        end

        in bank let getAc = proc(  accountNumber : int ;
                                   passwd : string → account )
        begin
                let new = lookup( accountNumber )
                if new( pin ) = passwd then new else failAc
        end
end
```

The important difference between this new procedure, *transfer*, and those already discussed is that it is defined over more than one object of the witness type. This causes no problem with the definition of the interface, as the type of the operands is declared prior to the type of the procedure. Although the type is abstracted over, the parameters are bound to the same definition, and so are restricted to being the same representation type. Similarly, at the place where the procedure is written (within the scope of the representation type), it is written over two objects of the same type. If it were not, a type-checking error would be detected in the attempt to create the abstract type.

This example illustrates a major difference in power between first-order and second-order information hiding. With second-order, a type is abstracted over, and procedures may be defined over this type. With first-order hiding, it is the object itself which is hidden within its procedural interface. Procedures which operate over more than one such object may not be sensibly defined within this interface. Therefore any operations defined over two instances must be written at a higher level, using the interface. At best this creates syntactic noise and is inefficient at execution time. It also means that such operations are defined in the module which uses the abstract objects, rather than the module which creates them. Some examples, such as this one, are not possible to write without changing the original interface.

This example highlights another difference between this style of abstract data type and that used in the language ML. Although the use of the type is similar to an ML-style type, the definition of structural type equivalence allows objects of the type to be passed between different compilation units. This is not the case with ML abstract types, which are only compatible if they refer to the same instance of the type definition: construction and use by independently-prepared modules is not possible by this mechanism and must be achieved otherwise [Har85].

6 Privileged data access and incremental system construction

We have seen how abstract data types may be used in conjunction with environments to provide a safe and flexible viewing mechanism, of which viewing over a database style relation is a special case. We will now show another paradigm, which allows privileged users to access the raw data without danger of losing the integrity of the abstract interfaces. This solves some traditional problems associated with abstract data types, and shows a way towards some of the more general problems associated with system evolution and incremental system construction.

A problem exists with abstract data types in long-lived persistent systems if, when the abstract type is formed, access to the original representation of the data is lost. This is a requirement of these types for the purposes of safety and abstraction, but there is an underlying assumption that no access to the data will ever be required apart from that specified in the abstract interfaces. For a large body of persistent data this is unrealistic.

A serious problem occurs if an error leaves the database in an inconsistent state. In the banking example, this could happen if one of the auto-tellers develops a mechanical failure which prevents it from dispensing the requested money after the *withdraw* operation has been executed. The easiest way for such an error to be rectified is for a privileged user, such as the database administrator, to be allowed access to the concrete representation of the account and adjust the balance. If such access is not allowed, then an error may occur which leaves the database in an inconsistent state from which it is not possible to recover.

This kind of access may be treated as another view over the same data, with no abstraction in the interface. It may be achieved using the environment mechanism, by keeping the data representations in a known place in the persistent store as shown earlier; this also allows suitable abstraction for the programs which manipulate the concrete data.

To prevent unauthorised users from gaining access to the information contained in this environment, a password protection scheme may be used. So that the type of the data can not be discovered from a scan of the containing environment, this environment can be hidden inside a procedure which requires a password:

```
let makeSecretEnv = proc(  password : string → proc( string → env ) )
begin
        let new = environment()

        proc( attempt : string → env )
                if attempt = password then new else fail
end

in dbAdmin let accountRepEnv = makeSecretEnv( "friend" )
```

Now the restricted definitions and data can be placed safely in this environment without fear of access by unprivileged users. This technique gives a result not dissimilar from the kind of module provided by Pebble [BL84] and a high level language analogy of capabilities [NW74].

This level of data access is also desirable for database programmers in other circumstances. It is not uncommon for a new operation to be required on data which is abstracted over: an example of this is if the procedure *transfer* described above became a requirement after the system had been installed. Although it may be possible to write such operations in a new user module, such use is often contrived and will always be inefficient. Using this technique, the database administrator may construct a new interface over the representation level of the data, with no associated loss of efficiency.

The above holds whether the new interface required is confined to adding an extra operation to one of the existing views, or whether an entire new view is necessary. The autoteller example given above could thus be added to an existing system, rather than being part of the design of the original system. Furthermore, none of the code would be any different from that shown, and programs which used the original interface would continue to work. This demonstrates the flexibility of this paradigm, and shows a way forward for the incremental construction of complex software systems.

7 Conclusions

We have demonstrated how the existentially quantified types of Napier88 may be used to provide a viewing mechanism over persistent data. Since one object may be a component of many existentially quantified objects, multiple views over the same data or database may be constructed.

We provide one example of constructing multiple views over a collection of bank accounts. Although we have not demonstrated it, it is possible to construct higher order views by imposing an existential interface over existing existential types.

We have also demonstrated that the technique for providing views and structuring the database can be used for data repair without compromising the abstract interfaces. Such a situation is similar to query processing in object-oriented databases where queries must have access to data in an object that is not available through the objects interface [BCD89].

The final attribute of the system is that the views are statically type checked.

As a practical demonstration of the technique, Appendix 1 is a listing of the code required to create and use one of the abstract interfaces described in the text. The appendix consists of three separate compilation units: the first creates the raw data and procedures and places them in a password-protected environment; the second constructs the abstract interface and places it in an unprotected environment; and the third shows an example of using the interface. The code has been fully tested in the Napier system.

8 Acknowledgements

We acknowledge discussions with Sandy Heiler and Stan Zdonik on viewing mechanisms in object-oriented databases which took place instead of some fluid dynamics experiments.

9 References

[ABC76] Astrahan M.M., Blasgen M.W., Chamberlin D.D., Eswaran K.P., Gray J.N., Griffiths P.P., King W.F., Lorie R.A., McJones P.R., Mehl J.W, Putzolu G.R., Traiger I.L., Waid B.W. & Watson V.
"System R: A Relational Approach to Data Management"
ACM TODS 1,2 (June 1976), pp 97-137.

[ABC83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An Approach to Persistent Programming"
Computer Journal 26,4 (November 1983), pp 360-365.

[ABM88] Atkinson, M.P., Buneman, O.P. & Morrison, R.
"Binding and Type Checking in Database Programming Languages"
Computer Journal. 31,2 (1988), pp 99-109.

[ACO85] Albano, A., Cardelli, L. & Orsini, R.
"Galileo : A Strongly Typed Conceptual Language"
ACM TODS 10,2 (June 1985), pp 230-260.

[BCD89] Bancilhon, F., Cluet, S. & Delobel, C.
"A Query Language for the O_2 Object-Oriented Database"
2nd International Workshop on Database Programming Languages, Salishan, Oregon (1989).

[BL84] Burstall, R. & Lampson, B.
"A Kernal Language for Abstract Data Types and Modules"
Proc. international symposium on the semantics of data types, Sophia-Antipolis, France (1984). In **Lecture Notes in Computer Science**. 173. Springer-Verlag (1984).

[CW85] Cardelli, L. & Wegner, P.
"On Understanding Types, Data Abstraction and Polymorphism"
ACM Computing Surveys 17,4 (December 1985), pp 471-523.

[Dea89] Dearle A.
"Environments : A Flexible Binding Mechanism to Support System Evolution"
Proc. Hawaii International Conference on System Sciences (1989), pp 46-55 .

[Har85] Harper R.
"Modules and Persistence in Standard ML"
Proc Persistence Data Types Workshop, Appin, Scotland (1985) in
Data Types and Persistence (Ed. Atkinson M.P., Buneman P. & Morrison R.) Springer-Verlag (1985), pp 21-30.

[IBM78] IMS/VS Publications
IBM, White Plains, N.Y. (1978).

[Mat85] Matthews, D.C.J.
"Poly Manual"
Technical Report 65, University of Cambridge, U.K. (1985).

[MBC88] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A.
"Napier88 Reference Manual"
Persistent Programming Research Report PPRR-77-89, University of St Andrews. (1989).

[Mil83] Milner, R.
"A proposal for standard ML"
Technical Report CSR-157-83. University of Edinburgh.

[MP85] Mitchell J.C. & Plotkin G.D.
"Abstract Types have Existential type"
ACM TOPLAS 10,3 (July 1988), pp 470-502.

[NW74] Needham R.M. & Walker R.D.
"Protection and Process management in the CAP computer"
Proc. International Workshop on Protection in Operating Systems. IRIA Rocquencourt, (August 1974), pp 155-160.

[Pow85] Powell M.S.
"Adding Programming Facilities to an Abstract Data Store"
Proc. Persistence Data Type Workshop, Appin, Scotland. Universities of Glasgow and St Andrews PPRR-16-85 (Ed. Atkinson M.P., Buneman P. & Morrison R.) (1985) pp139-160.

[ps87] "The PS-algol Reference Manual fourth edition", Universities of Glasgow and St.Andrews PPRR-12 (1987).

[SWK76] Stonebreaker M., Wong E., Kreps P, & Held G.
"The Design and Implementation of INGRES"
ACM TODS 1,3 (1976) pp 189-222.

```
!** Appendix 1 Unit 1.  Creating the concrete data types and procedures

!** Construct the concrete data type and procedures to work over it **

type account is structure( balance,Limit : int ; pin : string )
  !* "limit" is a reserved word for use with raster graphics

let failAc = account( 0,0,"" )
let lookup =       !** dummy for "lookup"
begin
     let myAc = account( 1,-1000,"friend" )
     proc( ac : int -> account ) ; if ac = 1234 then myAc else failAc
end

let getAc = proc( accountNumber : int ; passwd : string -> account )
begin
     let new = lookup( accountNumber )
     if new( pin ) = passwd then new else failAc
end

let withdraw = proc( debit : int ; ac : account )
begin
     let result = ac( balance ) - debit
     if result > ac( Limit ) and debit > 0 do ac( balance ) := result
end

let balance = proc( ac : account -> int ) ; ac( balance )
let Limit = proc( ac : account -> int ) ; ac( Limit )

!** Place the procedures in a new environment

let bankrep = environment()
in bankrep let failAc = failAc
in bankrep let withdraw = withdraw
in bankrep let balance = balance
in bankrep let Limit = Limit
in bankrep let sufficient = sufficient
in bankrep let getAc = getAc

!** Read a password for this environment from database administrator

let bank = environment()

let password =
begin
     writeString( "Enter bank admin password:'n" )
     readString()
end

in bank let retrieveBankRep = proc( s : string -> env )
     if s = password then bankrep else environment()

in PS() let bank = bank
```

```
!** Appendix 1 Unit 2.  Creating the abstract interface

!** Construct the abstract type definitions **

type amount is int      ! an amount of money
type number is int      ! an account number
type passwd is string    ! a secret code

type localTeller is abstype[ ac ]
(
     failAc   : ac ;
     getAc    : proc( number,passwd -> ac ) ;
     withdraw : proc( amount,ac ) ;
     balance  : proc( ac -> amount ) ;
     Limit    : proc( ac -> amount )
)

!** Get the representation environment from the store

let bankrep = use PS() with bank : env in
              use bank with retrieveBankRep : proc( string -> env ) in
begin
     let password =
     begin
          writeString( "Enter bank admin password:'n" )
          readString()
     end

     retrieveBankRep( password )
end

!** Create a new environment for the tellers and place it in the store **

let tellerEnv = environment()
use PS() with bank : env in in bank let tellerEnv = tellerEnv

!** Construct the abstract objects **

type account is structure( balance,Limit : int ; pin : string )
use bankrep with
     failAc : account ;
     withdraw : proc( int,account ) ;
     balance,Limit  : proc( account -> int ) ;
     getAc : proc( int,string -> account ) in
begin
     in tellerEnv let local = localTeller[ account ](
                                                     failAc,
                                                     getAc,
                                                     withdraw,
                                                     balance,
                                                     Limit
                                                  )
     writeString( "Teller interface successfully placed in tellerEnv.'n" )
end
```

```
!** Appendix 1 Unit 3.  Example of using the abstract data type

!** Construct the abstract type definition **

type amount is int       ! an amount of money
type number is int       ! an account number
type passwd is string    ! a secret code

type localTeller is abstype[ ac ]
(
     failAc   : ac ;
     getAc    : proc( number,passwd -> ac ) ;
     withdraw : proc( amount,ac ) ;
     balance  : proc( ac -> amount ) ;
     Limit    : proc( ac -> amount )
)


use use PS() with bank : env in
    use bank with tellerEnv : env in
    use tellerEnv with local : localTeller in local
as package in
begin
     let failAc = package( failAc )
     let getAc = package( getAc )
     let withdraw = package( withdraw )
     let balance = package( balance )
     let Limit = package( Limit )

     writeString( "Please enter your secret code:'n" )
     let code = readString()
     let myAc = getAc( 1234,code )

     if myAc = failAc then writeString( "Incorrect password'n" ) else
     begin
          writeString( "Your balance is " )
          writeInt( balance( myAc ) )
          writeString( ".'n" )

          withdraw( 10,myAc )
          writeString( "I have withdrawn 10 pounds.'n" )
     end
end
```