This paper should be referenced as:

Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Dearle, A., Rosenberg, J. & Stemple, D. “Protection in Persistent Object Systems”. In **Security and Persistence**, Rosenberg, J. & Keedy, J.L. (ed), Springer-Verlag (1990) pp 48-66.

Protection in Persistent Object Systems

Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A.†,
Kirby, G., Rosenberg, J.‡ and Stemple, D. ¥

University of St Andrews, Scotland
† University of Adelaide, Australia
‡ University of Newcastle, Australia
¥ University of Massachusetts, USA

Abstract

Persistent programming is concerned with the creation and manipulation of data with arbitrary lifetimes. This data is often valuable and therefore protected to ensure that it is free from misuse. The mechanisms used to protect the data vary with a tradeoff between static expression of the protection and the flexibility in modelling it. In this paper we explore the full range of protection mechanisms in persistent systems from static to dynamic checking and contrast it with the corresponding balance between safety and flexibility in the system. Protection by capabilities, dynamic universal union types, encapsulation, subtype inheritance, existential quantification and predicate defined invariants will be explored with reference to manipulating long lived data.

1. Introduction

Persistent object systems support large collections of data that have often been constructed incrementally by a community of users [ABC83]. Such data is inherently valuable and requires protection from deliberate or accidental misuse. Protection is required to guard against system malfunction, such as hardware failure, to ensure that users do not misuse the common facilities such as the operating system and finally to protect users from other users and even themselves.

Before discussing methods of enforcement we will place protection in the context of the total system function. Protection mechanisms are concerned with conserving the integrity of data. There are two quite separate ways in which this integrity may be compromised. Firstly, some kinds of failure, which jeopardise this integrity, such as hardware malfunction, have little to do with protection and are best dealt with by other techniques such as incremental dumping or stability strategies [Lor77,RHB90]. That is, periodically the data is copied to a secure device from which it may be retrieved if a failure occurs. Failures may, of course, occur in all the multiple copies of the data simultaneously leading to the conclusion that there is no absolute notion of data integrity. Protection mechanisms are built on the assumption that the underlying system does not fail. In the case of hardware error this assumption is incorrect. Fortunately an acceptable level of stability can usually be obtained for an acceptable price but the notion of absolute integrity no longer exists.

Given an acceptable level of stability, the integrity of the data can still be compromised by users. A number of mechanisms, such as capability systems, encryption methods, type systems and database integrity constraints have been used to add protection against data misuse in persistent systems. All of these mechanisms add security but at some cost and furthermore there is also a limit to the level of security that can be achieved. For example, once a program gains access to data it may alter it in a legal but undesirable manner. This can happen because the constraints placed on the user by the capability system, type system or integrity constraint system provide rather coarse grain control over the data. This coarse grain control trades expressiveness for security. The principle of minimum necessary privilege often causes the protection mechanism to be too fine grained to be enforced efficiently or even expressed succinctly. For example, the granularity of most type systems allows the user to restrict the use

of a variable to a particular type such as integer. A finer grained system may allow the specification of types that only allow a subrange of the values such as 3..4 on Tuesday and Wednesday and 2..5 on other days. This is fine grained but neither succinct nor easily efficiently enforced.

Thus the second problem with integrity is that the checking system may allow the integrity of the data to be preserved but still contain information that is no longer of real use or is even erroneous. That is, even when enforced, the protection mechanism will not guarantee that programs produce desirable transformations on the data. For example, the age recorded for a particular person may be incorrect. The challenge is to find an acceptable level of security that can be obtained for a reasonable price in terms of efficient checking and succinct expression.

In protection mechanisms there is always a tension between the time of checking and the flexibility of the system. Checks that can be performed by static analysis allow the programmer to state or even prove some properties of the program before it runs. This increases confidence that the program is correct and explains the desire of most programming language designers to employ static type checking as one of the mechanisms for protection. The same desire has also led to capability systems that can be statically checked [JL78] and to proposals for the static checking of database integrity constraints [SS89] A second aspect of static checking is that programs so checked are usually more efficient. By performing the checking statically the need for dynamic checking is reduced making the program execute faster and in less space.

Taken to the extreme statically checked languages are not very interesting since they have a very limited ability to accommodate change. Flexibility is introduced by dynamic checking. Constraints placed on the data that depend on the evaluation of the program may only be checked in this manner. For example, the integrity constraints checked just prior to a transaction commit usually employ dynamic checks of this kind.

Another dimension of static versus dynamic checking is the ability of the system to support its own evolution safely. This property is sometimes called reflection [SFS90] and on other occasions type magic [AM87]. In capability systems, which are usually the most dynamic, the type magic is the ability to issue and check the validity of capabilities. Capabilities are only issued by a trusted kernel and may not be manufactured by a user program. Compilers may generate representations of programs which when given a capability by the kernel become programs within the system. Since the security checking is always performed by the kernel, programs may be generated by many different compilers. Thus mixed language facilities are available up to the constraints imposed by the capability system. An example of such a system is given in [RA85].

As the checking is made more static the ability to mix languages is lessened. The magic in the system now lies with the type checkers which can only check languages with related type systems. The reflection in the system is also restricted by the type systems since only languages that adhere to the type system can be checked by the type checker. This is usually less general than a capability system. Example of systems displaying high level reflection at the type system level are [PS87,MBC88].

The semantics of failure is another dimension of protection that depends upon the time of checking. With dynamically checked systems an error such as an invalid capability or type error may be discovered at run time. An exception mechanism may be used to deal with such errors. In a statically checked system the errors are discovered at compile time and there is no notion of an exceptional event for these checks.

One final point is that the system security can be compromised if the type magic can be impersonated. This is often necessary for maintenance and repair of systems and again yields a weakness in the security system. It often depends on how difficult it is to break the password checking for initial access to the system.

We will now explore the full range of protection mechanisms in persistent systems from static to dynamic checking with the corresponding balance between safety and flexibility. Protection

by capabilities, dynamic universal union types, encapsulation, subtype inheritance, existential quantification, and predicate defined invariants will be explored with reference to manipulating long-lived data.

2. Capabilities and Type Systems

Capabilities were first proposed by Dennis and Van Horn [DH66] as a technique for describing the semantics of controlled access to data. The idea was extended by Fabry who proposed a computer system based on capabilities [Fab74]. There have been several attempts at constructing such a capability-based system. Some of these enlisted hardware support and others were purely software implementations. Although these systems differ greatly the fundamental principles of capability-based addressing are the same.

The basic idea is that access to objects is controlled by the ownership and presentation of capabilities. That is, in order for a program to access an object it must possess a capability for the object. In this sense capabilities may be viewed as keys which unlock the object to which they refer. Since the possession of a capability gives a right to access an object it is important that programs are not able to manufacture capabilities, since this would allow a program to access data which was not supposed to be available to it. Methods of protecting capabilities include segregation [WLH81,WN79], tagging [Feu73,MB80] and password schemes [APW86]. A capability for an object can thus only be obtained by creating a new object or by receiving a capability from another program.

Capabilities have three components. These are a unique name identifying the object, a set of access rights and some status information as shown in Figure 1. Capability systems use names for objects which are unique for the life of the system. The name given to an object will never be re-used, even if the object is deleted. This avoids aliasing problems and provides a means of trapping dangling references. Such unique names are not difficult to generate and addresses in the order of 64 bits are sufficient to ensure that the system will never exhaust all possible names.

Figure 1 : A Capability Structure

Although the ownership of a capability guarantees the right to access the corresponding object, the access rights field may restrict the level of access allowed. The facilities provided by access rights vary greatly between different capability systems. They may be as simple as read, write and execute, or they may be based on the semantics of the different objects, for example a list of procedures for accessing an abstract data type. When a capability is presented in order to access an object, the system checks that the type of access does not exceed that specified in the capability. There is usually an operation which allows a new capability to be created from an existing one with a subset of the access rights allowing for the construction of restricted views.

The third field of a capability contains status information which indicate which operations can be performed on the capability itself. Again, these vary greatly. The minimum usually provided is a *no copy* bit which restricts the copying of the capability, perhaps on a user basis. This may be used to stop a user from passing a capability on to other users to limit propagation. Other status bits may include a *delete* bit which allows the holder of the capability to delete the object.

A final facility provided on some capability systems is the ability to revoke access. That is, after giving a program a capability it may be desirable at a later time to revoke this capability. Implementation of revocation may not be easy. The simplest technique is to change the unique name of the object which will effectively invalidate all existing capabilities. Selective revocation may be supported by using indirection through an owner controlled table of access rights or by providing multiple names for the object which can be individually invalidated.

Capabilities provide a uniform model for controlling access of data. However, entry to the system itself, by logging on, must in the end be based on some form of password. An advantage of capability-based systems is that, even if the password system is broken, there need not be any single password which provides access to all data of the system. That is, there need not be a superuser.

Capability systems employ dynamic checking and represent one end of our spectrum of checking times. Being dynamic they are expensive to implement since the check has to be performed every time the capability is used. For this reason hardware assistance is usually employed for efficient implementation. The expense of implementation partially explains why capabilities have commonly been used to control access to data via the addressing mechanism rather than to perform more elaborate dynamic type checking. Validating capabilities usually means comparing two addresses which is a cheap operation that may be implemented efficiently.

The dynamic nature of the checking means that a mechanism must be provided to allow recovery from checking failure. An exception handler is one such mechanism. This semantics of failure allows more flexibility in the system since capabilities can be revoked or altered dynamically without fear of violating the protection mechanism. Furthermore, a capability system is required to arbitrate among programs and data that mutually distrust one another. This distrust may change as the programs are evaluated and eventually wish to communicate. A capability system is able to deal with these changing needs.

Controlling access to objects in a capability system is achieved by limiting a program to a subset of the access rights of the object. Capabilities address objects with these access rights. An analogy has been made with abstract data types where the interface procedures define the access rights [JL78]. Limiting the interface procedures by a viewing mechanism or a subtyping mechanism allows the restrictions on access normally found in capability systems to be extended to type systems.

This analogy also allows an extension of more traditional capability systems. Checking for equality of capabilities depends upon the system being able to generate unique addresses for each different access type. This may require a large number of address bits. Alternatively the checker can employ a more elaborate check to decide the equality of two capabilities; this could be a full type check. The difference is similar to the difference between name and structural type equivalence in type systems.

Type systems themselves provide two facilities within programming systems. These are a modelling ability and a protection mechanism. Type systems may be dynamically checked but some systems employ static checking. In such systems there is no notion of type failure at run time and that the checking may be factored out at compile time.

It has long been realised that in supporting independently prepared program and data in a persistent store, some form of dynamic checking is required to perform the binding [ABM88]. That is, where a program is to be bound to data expressed at run time, the system is required to support this dynamic bind. Part of this binding involves a check that the type is correct [MBD88]. Thus some dynamic checking is performed and has many similarities to software capabilities.

It is interesting to speculate how and where dynamic checking of a capability system may be made static. It is also interesting to discover how and where statically typed persistent data requires dynamic binding. We explore both of these issues in the following sections.

3. Dynamic Type Checking

Capabilities may be implemented in software and even within type systems. The cost of software capability checking is high and therefore it is sensible to use it less often than where hardware checking would be used. Examples of software capabilities are dynamic integrity checks that occur just prior to a transaction commit and in the projections out of infinite union

types found in some programming languages. In the first case the capability being checked is the capability to allow the transaction to proceed. This capability is very flexible and depends on the values within the database itself. In the second case, the system is checking that the user has the correct type with which to use an object.

The difficulty with capabilities described by dynamic types is that all computations in the system have to be couched in terms of the type algebra. This is usually where the ability for mixed languages is lost since the system can only describe computations in languages whose type systems are subsets of the overall one. Furthermore the type systems only provide controlled access to objects and not protection on how the capability is itself manipulated. There are usually no type system equivalent operations for *no copy*, *revoke* and *delete*. To implement such facilities another layer of abstraction must be placed on the capability leading to further inefficiency.

The difference between dynamically checked type systems and capabilities is in the algebra used to describe access in capability systems, and therefore the ability to manipulate the object, and the algebras over types. Capability descriptions are usually more flexible and extensible than type algebras although this need not be the case. As described above, the capability can be considered as the access route to an object and the view of the data accessible from that capability. To obtain two different types of access requires two different capabilities. This is also true in type systems with the difference being the mechanism used to construct the view.

Type **any** in the languages Amber [Car85] and Napier88 [MBC88] is an infinite union type that can contain an object of any type. Type checking is performed on the type at the point of dynamic projection from the union. An object may be injected into type **any** where it is then compatible with any other object of type **any**. For specific use the object must be projected from type **any** onto its exact type. The following example shows how an integer with the operations read and write may have its access restricted to reading.

```
type intAccess is structure (read : proc (→ int) ; write : proc (int))
type intReadAccess is structure (read : proc (→ int))

let protectedInt := 0
let Read = proc (→ int) ; protectedInt
let Write = proc (x : int) ; protectedInt := x

let rwobject = intAccess (Read, Write)
let robject = intReadAccess (Read)

let readCapability = any (robject)
let readwriteCapability = any (rwobject)
```

Figure 2 : Read and Read Write capabilities

In Figure 2, two structure types are declared. The first, *intAccess*, contains two fields containing the procedures, *read* and *write* and the second, *intReadAccess*, a procedure field called *read*. The declaration of *rwobject* initialises it to an instance of the type *intAccess* which contains the procedures *Read* and *Write*. These procedures manipulate the object *protectedInt* within their closures.

The object *robject* is initialised to a structure with the same read procedure as the other structure *rwobject*. The objects *robject* and *rwobject* have different types for static type checking. However, by injecting them into the infinite union **any** they now have the same universal type. Thus, *readCapability* and *readwriteCapability* can be passed around interchangeably. Just like proper capabilities they are not equal to one another. Notice that in the above example only the access mechanism is protected not the original data. This could be remedied by using a procedure with a password parameter to generate it as can be seen later. Figure 3 shows how the capabilities may be used.

```
let useobject =     proc (capability : any)
                  project capability onto X as
                      intAccess          : ... ! code using read and write
                      intReadAccess      : ... ! code using read only
                  default                : ... ! code for some other activity

useobject (readCapability)
useobject (readwriteCapability)
```

Figure 3 : Using Capabilities

In Figure 3, the procedure *useobject* takes an object called *capability* of type **any** as a parameter. The **project** clause allows the user to match the type against the real one. If a match is found then the original interface is exposed. No coercion or breaking of the type system is allowed, merely a dynamic check.

Such a type facility has many similarities to capabilities in that the objects of the universal union type can be used interchangeably, and expose only the correct interface to an object on projection. Original data can also be protected by password. More controversially the mechanism cannot be used directly to revoke an access right. The user of such a dynamic type must state statically the access they are to be allowed. If the dynamic test of these rights succeeds they are guaranteed for as long as the user expects to possess the object.

Infinite union types allow partial specification of the schema in persistent systems. That is, to use the persistent store the type must be described at least to the points where dynamic resolution will take place. Type **any** described above can be used for this and the facility allows incremental evolution of the system since the partial specification can always be extended by a new object of type **any** with a different real type. Where programs and data are prepared separately as in persistent systems there is a requirement to protect the dynamic binding of them. This involves dynamic protection of the kind described with type **any**.

One final variation on software capabilities is the module mechanism of the language Pebble [BL84] where the modules may be protected by a password. Access to a module depends upon calling it with the correct password which is similar to protection in most capability systems except that the passwords are explicit. Different access to the same object is achieved by using different passwords.

All of these methods of software capabilities simulate the access control aspects of capability systems. None of them controls the distribution of capabilities themselves. This has to be done by a separate mechanism.

Given the requirement for capabilities in persistent systems, even if implemented by software, we now turn our attention to what protection can be implemented statically.

4. Statically Checked Capabilities

Jones & Liskov [JL78] have proposed a system where the checking of access is performed statically. In this, each object has a type which determines the legal accesses to it. Variables are used to access objects and are declared to have a subset of the full access rights. Variables have qualified types which have two parts. A qualified type Q is written $T\{r_1,\ldots r_n\}$, where T is some type name and $\{r_1,\ldots r_n\}$ a subset of the access rights. The two parts of the qualified type are the base, where base (Q) = T and the access rights, where rights (Q) = $\{r_1,\ldots r_n\}$.

The variables contain capabilities since each different variable potentially provides a different set of access rights. The problem now is to find a substitution rule where capabilities may be substituted for one another, that is assigned or passed as parameters, while conserving static checking.

The substitution,

$v \leftarrow e$

where v is a variable of type T_v and e is an expression of type T_e is valid if,

$Te \geq Tv$

Type T_e is greater than or equal to T_v if,

base (T_e) = base (T_v) and rights (T_e) ≥ (T_v)

That is, a substitution is valid if the new access path provides at most a subset of the old rights. An example given by Liskov and Jones is the procedure heading,

procedure P (x : T1 {f,g}) **returns** T2 {k}

and the declarations,

a : T1 {f,g,h}
b : T2 {k}
c : T1 {f,h}

The statement b ← P(a) is legal and statically checkable since $T_x \leq T_a$ in the call and T_b is the same as the return type T2{k}. However b ← P(c) is not legal since $T_x \nleq T_c$. The reason for disallowing this is that the procedure may make use of the access right *g* specified in its interface and that it is not available from the calling parameter expression.

This notion of type is a simplified version of what has more recently become known as inclusion polymorphism [CW85] in that T_e is considered as a subtype of T_v for the binding to be valid. Apart from the inequality symbol being reversed the semantics are the same.

Jones and Liskov also found the semantic anomaly later described in [Car89,AGO89] for structured types containing references. Consider their example,

procedure P (a : **array** [T {f}]{**all**} ; x : T {f})
begin
    a [1] := x
end ;

The array declaration specifies that the elements are restricted to T{f} and that all the operations on arrays are available. The procedure appears to be type correct since in the assignment the value *x* has the same type as the elements of the array, namely T{f}. However, consider the declarations,

b : **array** [T{f,g}] {**all**}
y : T {f}
z : T {g}

and the call

P (b,y)

This call also appears correct since $T_b \leq T_a$, since rights (T_b) ≥ rights (T_a), and $T_y \leq T_x$. The problem is with the assignment,

z := b (1)

The value b (1) which is updated in the procedure to the value *y* has by this assignment the access right *f* only. The variable *z* has the access right *g* only. However the array has access

rights *f* and *g* and the assignment should be legal since it would have been had it been performed before the procedure call. This anomaly is not statically checkable.

To overcome this anomaly and preserve static type checking Jones and Liskov propose *?types* which are now more commonly known as bounded universal quantifiers. The substitution rule for objects of this type is the same as proposed in Napier88 to overcome the same subtyping difficulty. That is, that two such types are only substitutable if they known to have exactly the same access rights. It should be noted that two types with different *?types* names but the same set of access rights cannot be guaranteed to have the exactly the same type.

For example the above would now be written as

```
procedure P [ t ≤ ⊤ {f}] (a : array [t] {all}; x : t)
begin
      a [1]:=(x)
end
```

The assignment is legal since the elements of a and x have the same type t. However the call

```
P (b,y)
```

is not legal since *t* would have to have rights {f,g} and {f} simultaneously. While either one is acceptable, both are not simultaneously. This rule also overcomes a simpler but similar semantic anomaly specified by Cardelli for inclusion polymorphism.

5. Information Hiding

A higher level method of protecting data within software systems is the use of information hiding. This is defined as any programming method which limits the computation allowed by the type system upon data, by restricting either the access or type interface to it. It may also be possible to provide abstractions over the basic operations defined by the type system, including complex and dynamically evaluated constructs. Where such abstractions have type system support they are usually referred to as abstract data types. It is important to note that information hiding may only be relied on as a protection mechanism within the context of a strong type system, since if arbitrary address arithmetic is allowed data may always be accessed from outside the programmed interface.

Programming language type systems themselves operate by the use of information hiding over the operating system and hardware operations available to them. This abstraction is at a lower level, and is not necessarily available to the programmer. For example the integer type is defined in most type systems not as a mathematical integer but instead as a restricted interface over its hardware implementation. In strongly-typed languages the user is prevented from using operations such as rotate and xor on these values, and extra functionality such as conversion to and from equivalent character strings is also provided. Similarly, in a system which does not use hardware capabilities, it is only the use of information hiding which may prevent the use of arithmetic on address values.

There are three well-known mechanisms which allow the programming of information hiding within a strong type system. These are subtype inheritance, procedural encapsulation (1st-order information hiding) and existential data types (2nd-order information hiding). Subtype inheritance achieves protection by removing type information, causing the static failure of programs which may try to perform undesirable accesses. 1st-order information hiding prevents the protected data from being named by an untrusted program, allowing access only through a procedural interface. 2nd-order hiding is somewhere between these two, allowing access mainly through procedures, but also allowing the protected data to be named. This data is, however, viewed through a mechanism which causes type information loss, which allows only a limited set of operations to be performed on it. These mechanisms may also be used effectively in combination.

5.1 Subtype Inheritance

In general, systems which allow subtype inheritance allow any data object to be used in place of one with less functionality. One type is a subtype of another if it defines all, and perhaps some more, of its operations. In the most general form of subtyping, known as inclusion polymorphism, it is type correct for the use of any value to be replaced by the use of any of its subtypes.

Subtype inheritance is usually viewed as a general modelling technique. In particular it allows the declaration of procedures which operate over any type with at least a set of required properties. However, using an object as one of its supertypes is also equivalent to hiding some of the functionality which the object possesses. For example, the following introduces the names *employee* and *person* as record types:

```
type employee is structure (name,address : string ; salary : int)
type person is structure (name,address : string)
```

In the above, two structure types are declared. The first, called *employee,* has three fields called *name, address* and *salary* with types string, string and integer respectively. The second, called *person,* has two fields called *name* and *address* both of type string. Type *employee* is a subtype of type *person* using implicit structural equivalence and an object of type *employee* may be used in any context where an object of type *person* is expected. This would have the effect of hiding the salary field by the loss of type information. If another user is only to be allowed this restricted access to employee objects, this view of the object may be exported, for example by use of an explicit type coercion:

```
let joe = employee ("Joe Doe","1 Assignment Boulevard",100000)
let exportJoe : person = joe
```

In this *joe* is declared to be an object of type *employee* with the given field values. *exportJoe* is of type *person,* denoted by the type after the : symbol, but has the value *joe.* This means that a user of the value *exportJoe* will now have the value of the original record. However, it is not possible to express an operation to access the salary field of this value due to the restrictions of the static type system. That is, the *salary* field cannot be used with the object *exportJoe* since such a program would not type check, even although it would be able to execute without error.

This mechanism allows only simple information hiding compared with the other methods of 1st and 2nd order information hiding. Its advantages are that it is simple and elegant to use, and is easy to understand.

This inheritance is more flexible than the scheme described by Jones and Liskov, which defines essentially the same subtyping rule but allows it only for values of a particular type. In their scheme, an object is created with a finite set of operations of which restricted views may be passed around. In this scheme arbitrary types may be used, so long as they contain the common functionality.

5.2 1st-Order Information Hiding

Access to data can also be restricted by only allowing access to procedures which are defined over the data, and not allowing the data itself to be visible. This is a common model for abstract data types, and is known as 1st-order information hiding. It may be achieved in a number of ways but we will describe it in terms of a language which has first-class procedure values and block-style scoping. Access to the original data may then be removed simply by its name becoming unavailable. This allows a much more flexible interface to be constructed. For example, the following type defines a *Person* as a record containing procedures which define three operations:

```
type Person is structure (getName, getAddress : proc (→ string) ; putAddress : proc
(string))
```

This allows a finer restriction than that above in that the name and address may be read, but only the address may be changed. Access to the data may be removed by placing its declaration in a block which is retracted immediately after the *Person* object has been constructed and exporting procedures with the data encapsulated within their closures, as shown in Figure 4. Again, this relies upon the static properties of the system to prevent the access since a program which attempts direct access to *joe* will fail statically.

```
let exportJoe =
begin
        let joe = employee ("Joe Doe","1 Assignment Boulevard",100000)
        Person (proc (→ string) ; joe (name),
                proc (→ string) ; joe (address),
                proc (new : string) ; joe (address) := new)
end
```

Figure 4 : Hiding the Data Representation

In Figure 4, *exportJoe* is declared to have the value obtained by executing the block. This is a structure of type *Person* with three procedure fields. Each procedure uses the object *joe* which is inaccessible by any other means on exit from the block. Notice that a number of different interfaces may be programmed like this, and exported from the original data, rather than just a single one as here. This allows the construction of multiple views on the same data.

Further flexibility is possible using encapsulation in that dynamic properties may be specified, and access may be denied dynamically if required. For example, perhaps there exists an integrity constraint that an address may not be more than 100 characters long. This can be programmed in the procedural encapsulation, as shown in Figure 5. The only difference here is that the *putAddress* procedure checks the dynamic constraint, and raises an exception if it is not met.

```
let exportJoe =
begin
        let joe = employee ("Joe Doe","1 Assignment Boulevard",100000)
        Person (proc (→ string) ; joe (name),
                proc (→ string) ; joe (address),
                proc (new : string) ;
                        if length (new) ≤ 100 then joe (address) := new
                                              else raise longAddress (new))
end
```

Figure 5 : Refining the Interface

A particular example of a dynamic constraint allows access to the original data to be protected by password, or by a software capability. A procedure can be provided in the interface which will return the access to a user with sufficient privilege. To prevent unrequired noise in this example we will use subtyping as above for users who do not expect to require this privilege, allowing most users not to even be aware of its existence. Figure 6 shows the extended definition required, with an extra procedure in type *extraPerson* which returns the raw data only if it is supplied with a string equivalent to the password used to create it. Now whoever is responsible for the construction of the view will have enough information to extract the representation. Alternatively, it is possible to arrange system-wide capabilities which would decide whether access is allowed or not.

```
type extraPerson is structure (getEmployee          : proc (string → employee) ;
                               getName,
                               getAddress    : proc (→ string) ;
                               putAddress    : proc (string))

let exportJoe = proc (password : string → extraPerson)
begin
      let joe = employee ("Joe Doe","1 Assignment Boulevard",100000)
      extraPerson (proc (s : string → employee)
                          if s = password then joe else failValue,
                   proc (→ string) ; joe (name),
                   proc (→ string) ; joe (address),
                   proc (new : string) ; joe (address) := new)
end
```

Figure 6 : Protection by Password

Since *extraPerson* is a subtype of *Person* an object of this type may be used where an object of the supertype *Person* is specified.

5.3 2nd-order Information Hiding

2nd-order information hiding does not restrict access to the protected values, but instead abstracts over the type of the protected value to restrict operations allowed on it. Thus the protected values may be manipulated for some basic operations, such as assignment and perhaps equality, but their normal operations are not allowed due to the type view. This allows the representation objects themselves to be safely placed in the interface along with the procedures which manipulate them.

This power can almost be achieved using a combination of subtyping and 1st-order hiding. For example, Figure 7 shows how a reference to the representation may be safely placed in the interface by effectively removing all type information from it. The representation may be accessed as a value, but only a highly restrictive access is possible as there is very little type information available. It may be assigned and tested for equality, but none of its fields may be accessed due to the static typing restrictions.

```
type Person is structure (absPerson          : structure ();
                          getName,
                          getAddress : proc (→ string);
                          putAddress : proc ( string))

let exportJoe =
begin
      let joe = employee ("Joe Doe","1 Assignment Boulevard",100000)
      Person (joe,
              proc (→ string) ; joe (name),
              proc (→ string) ; joe (address),
              proc (new : string) ; joe (address) := new)
end
```

Figure 7 : Removing the Type Information

In Figure 7, within the block *joe* is used to initialise one of the fields of type structure (). *joe* is of type *employee*, which is a subtype of structure () and the initialisation is legal. However, the fields of *joe* may not be accessed by this route.

Using this technique it is not possible to know that two such abstracted values are the same type, which may be desirable for some applications. For example, a *Person* may also have a

father and mother in the interface, along with a field for a favourite parent which changes between them. This technique does not provide enough information to allow this. For example, if the definition is:

```
type Person is structure (absPerson,mum,dad,favourite      : structure () ;
                          getName,getAddress               : proc (→ string) ;
                          putAddress                       : proc (string))
```

then it is not in general allowable to write

```
exportJoe (favourite) := exportJoe (mum)
```

as there is no way of telling that *mum* and *favourite* are indeed the same type.

A mechanism which allows 2nd-order information hiding is the existential data type described by Mitchell & Plotkin [MP88]. This allows the definition of interface types which are abstracted over. As names for these types are declared before the existential type definition, different parts of the definition may be bound to the same type. As before, only the basic operations defined on all types are allowed over these, but values which are abstracted by the same name are statically known to be compatible. *Person* as above may be redefined as:

```
type Person is existentialType [ absPersonType ] (
    absPerson,mum,dad,favourite   : absPersonType ;
    getName,getAddress            : proc (→ string) ;
    putAddress                    : proc (string))
```

The name in square brackets before the body of the type declaration declares a name for a type which is abstracted over. This allows a tighter definition of such types, as it can now be seen where the same type appears in the interface. For example,

```
exportJoe (favourite) := exportJoe (mum)
```

may now be statically determined to be type correct, as the *favourite* and *mum* fields must be type compatible to allow the object to be created.

This static binding of equivalent types may also be used to allow the interface procedures to be defined over the type of the hidden representation. A more flexible definition which allows the name and address operations to be performed on any of the people in the interface would be:

```
type Person is existentialType [ absPersonType ] (
    absPerson,mum,dad,favourite   : absPersonType ;
    getName,getAddress            : proc (absPersonType → string) ;
    putAddress                    : proc (absPersonType,string))
```

This allows the definition of n-ary operations over the hidden representation type. For example, a procedure may be placed in the interface which tests if two people have the same address:

```
type Person is existentialType [ absPersonType ] (
    absPerson,mum,dad,favourite   : absPersonType ;
    getName,getAddress            : proc (absPersonType → string) ;
    putAddress                    : proc (absPersonType,string) ;
    sameAddress                   : proc (absPersonType,absPersonType → bool))
```

This example illustrates a major difference in power between 1st-order and 2nd-order information hiding. With 2nd-order, a type is abstracted over, and procedures may be defined over this type. With 1st-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 defined sensibly 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 are not possible to write without changing the original interface. The power of such existential types is discussed fully in [CDM90].

6. Database Constraints

Capabilities and type-oriented protection are mechanisms based on interfaces, signatures, or access rights to objects. Another approach to protection is based on predicates stating invariants that must hold over changes to persistent objects. The simplest example of this is probably the sub-range types from most programming languages, e.g. ADA's subtypes with range constraints. Such types lead to notoriously difficult type-checking problems, which are normally resolved by dynamic checking. In databases, such protection is the domain of integrity constraint maintenance and can involve both static and dynamic checking.

The basic idea is quite simple: any type can be refined by adding a predicate on values of the type; only values obeying the predicate are in the new subtype. In databases, functional dependencies and referential integrity can be expressed in this way. A functional dependency is a predicate added to a relational type. Referential integrity is captured as a predicate stating that a column of one relation (a "foreign key" column) is contained in a key column of another (or the same) relation. This predicate, like other interrelational constraints, must be added to the database type itself. The problem is that non-trivial theorem proving is now required as part of static type-checking in order to avoid very expensive dynamic checks, and for quite simple predicate and manipulation languages static type-checking is either computationallyintractable or undecidable. One response to this difficulty is to limit the constraint and manipulation languages as well as raising their level of abstraction. With limited languages some effective theorems can be used to design procedures for checking computations and generating optimized run-time checks [BB81,BBC80,HI85,MH89,SS89]. Redundant data and special run-time integrity subsystems can be used to speed up checking. Most approaches in the literature work independently of a type system.

It is possible to integrate a theorem prover into the type checker in order to maximize the amount of static type checking achievable in the presence of predicates. The setting in which this appears to be feasible consists of high level languages limited in expressiveness and with the same formal base for the predicate and update languages. It is also possible that static predicative type checking will only be effective with small programs such as typical teleprocessing database transactions.

One benefit of a theorem prover embedded in a type checker is that a broader range of conditions could be used in specifying bounded universal quantification. Normally the conditions on the instantiating types of bounded universally quantified types only specify the existence of operators. This is easy to check. If a theorem prover is available, further predicates on the operators can be added. These conditions need to be proved from the properties of any instantiating type. This can be expensive and even incomplete, though it typically needs to be done in response to a compile-time resolvable declaration, not a run-time action.

Significant effort has gone into building an efficient, though necessarily incomplete, type checker for the set-oriented database programming (or specification) language ADABTPL [SS89]. Efficient proofs of integrity constraint maintenance have been achieved in ADABTPL by limiting the set of constraints and update primitives and by building a set of generic theorems about the interaction of the primitives.

Many of the ADABTPL theorems are higher order theorems, and engineering effective ways to use them is crucial to achieving efficiency. It is quite difficult to characterize the limits of the current ADABTPL techniques. It is, in effect, an expert system with heuristics coded in Lisp and rules that are all proved theorems. It can be improved by adding to its heuristics or to its theorems. The incompleteness of the reasoning is one of the aspects that is most troubling about this approach. Failing to prove that predicates are not maintained is ambiguous in this setting. It means either that the program can produce invalid data or it is too difficult to prove that it obeys the predicates.

There are several responses to a failed proof. The first is to add the unproved residue of the theorem to the transaction as a precondition. In many cases this is the proper response, indicating that the designer forgot the precondition. In other cases, the residue indicates that certain updates were left out. A simple case illustrates this. Suppose a database includes employees and dependents, and each dependent is constrained to relate to an employee. If we attempt to prove that a transaction consisting solely of a delete of an employee preserves the constraint, then an unprovable residue will be left at the end of the unsuccessful try. The residue will be the predicate stating that there are no dependents related to the employee being deleted. This predicate could be added as a precondition of the transaction, indicating that the transaction was only intended for use in deleting employees with no dependents. However, it may be the case that the designer forgot to include the deletion of the dependents in the transaction. In this case, the transaction is rewritten to include the deletions and the proof is attempted on the new transaction.

Because of the incompleteness of non-trivial, efficient theorem proving, the residue may in fact be true in all valid databases, but unprovable by the static checker. There are three ways of handling this case open to a designer.

- Allow the check to be made anyway, since the informal proof that convinced the designer that the residue was a theorem may be faulty.
- Simplify the transaction or integrity constraints specification without changing their meaning or change the transaction or constraints if consideration of the precondition reveals that they did not have the intended semantics and retry the proof.
- Tell the system that the check need not be made, thus overriding the checking in this case.

A description of the kind of theorem proving that is needed in assuring the maintenance of predicate-based integrity will clarify some of its limits and costs. The following theorem is the kind that needs to be proved to show that a transaction T leaves a database in a consistent state if it was consistent before the transaction. IC is the single predicate collecting all the different predicates constituting the specification of database integrity. I stands for the input to the transaction. (The input needs to be checked to see that it obeys its type constraints. This is a dynamic check, of course.)

$$IC(DB) \rightarrow IC(T(DB,I))$$

A form of theorem that is quite useful in proving such theorems is the following:

$$CONSTR(UPDATE(D)) = SIMPLER(D) \;\&\; CONSTR(D)$$

The idea behind this form is that the integrity of an updated database will be expressed in terms of constraint predicates (CONSTR) on updated data elements (D), the lefthand side of the equation. The righthand side form, which will replace expressions like the lefthand side during a proof, is a good form since the latter part (CONSTR(D)) will occur in the antecedent of the theorem, expressing the assumption that the database is initially consistent. This allows CONSTR(D) to be removed immediately from the proof obligation. Thus we are left with SIMPLER(D) where CONSTR(UPDATE(D)) appears in the theorem. An example will make this clear.

A theorem (assuming i is not in R) that is useful in maintaining a key constraint on a relation is:

$$key(insert(i,R),K) = i.K \notin project(R,K) \;\&\; key(R,K)$$

This states that the condition of a column K being a key of a relation R after a new tuple i has been inserted is equal to the K component of the new tuple not being in the K projection of R and R being keyed on K. This is useful in proving that a particular insert does not violate the key constraint. Since the second term in the righthand side is assumed to be true, the proof of the key property in the updated relation has been reduced to a check for non-membership in a

projection. Further proving may verify that the insert only occurs in a position in which the non-membership is assured, in which case this part of the overall theorem has been proven. If the non-membership cannot be proven, it can be added to the transaction as a dynamic check. This case is quite simple but indicates the basic approach to using theorem proving to assure that predicates are invariants of transactions.

There are two further points to be made. The first is that the SIMPLER predicate should be better than the constraint on the updated data in some sense, either by having less computational complexity or in facilitating further reduction, in order for this approach to be effective. The second point is that a uniform formal base and limitations on the integrity constraints and update functions are probably necessary in order to build a set of effective theorems (and heuristics) for this application of theorem proving. This approach embodies both static checking, in the form of the theorem proving, and dynamic checking, in the form of the reduced integrity constraints executed at run-time. Other than the cost of the theorem proving, which in ADABTPL currently runs to less than half a minute for six hundred term theorems on modern workstations, and the cost of the run-time checks, this approach requires building effective theorem bases for different styles of databases, limiting language expressiveness, and formalizing constraint and update languages in a uniform manner.

7. Conclusions

We have presented a number of mechanisms that are used to preserve the integrity of data in persistent systems. In statically checked systems, these mechanisms range from the integrity constraints of the language ADATPBL, through the statically enforced existentially quantified types of Napier88, encapsulation by scoping, subtype inheritance and the software capabilities of Jones & Liskov. In dynamically checked systems, the universal union type **any** of Amber and Napier88 may be used as software capabilities as may dynamically checked database integrity constraints. Hardware enforced capability systems form the most primitive but most efficiently implemented form of the techniques.

A major goal in system design is the ability to describe the properties of a system without having to execute it. However, as we have seen, this ability conflicts with the flexibility of dynamic checking. At any level of abstraction, the requirement is for static checking that is commensurate with the expressiveness needed for a particular application.

Not all constraints on data may be captured statically. Employing a theorem prover allows more powerful static checking but it may not be possible to prove all the theorems. This could be because they are in error or because the theorem prover is not sophisticated enough computationally to prove them in a reasonable time. A solution is to provide a more dynamic check where this occurs in a system.

A strategy, similar to the ones found in the ADAPTBL and DIADA [SWB89] projects, may be developed where all the constraints on data are specified in the one language. A theorem prover is then employed to capture as many of these constraints as it can statically. Where it cannot, a second, but less expensive in terms on technology, level of checking may be provided by a type system. Where the type system cannot enforce the constraint statically, a dynamic check is produced. This check may be enforced by hardware or software.

ACKNOWLEDGEMENTS

This work was undertaken during a period of study leave by John Rosenberg at the University of St Andrews which was supported by SERC grant GR/F 28571 and a visit to St Andrews by David Stemple.

REFERENCES

[ABC83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R."An Approach to Persistent Programming". *The 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". *The Computer Journal*. 31,2 (1988), pp. 99-109.

[AM87] Atkinson, M.P. & Morrison, R. "Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store". *2nd International Workshop on Persistent Object Systems*, Appin, (August 1987), pp. 1-12.

[APW86] Anderson, M., Pose, R.D. & Wallace, C.S. "A Password-Capability System", *The Computer Journal*, 29, 1, (1986), pp. 1-8.

[BB81] Bernstein, P. A. & Blaustein, B. T. "A Simplification Algorithm for Integrity Assertions and Concrete Views". *Proc. of the Fifth International Computer Software and Applications Conference*, (1981), pp. 90-99.

[BBC80] Bernstein, P. A., Blaustein, B. T, & Clarke, E. M. "Fast Maintenance of Semantic Assertions Using Redundant Aggregate Data". *Proc. of the Sixth International Conference on Very Large Databases*, (1980), pp. 126-136.

[BL84] Burstall, R. & Lampson, B. "A Kernel 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).

[Car85] Cardelli, L. *Amber*. Tech. Report AT7T. Bell Labs. Murray Hill, U.S.A. (1985).

[Car89] Cardelli, L. "Typeful Programming". *DEC SRC Report*, (May 1989).

[CDM90] Connor, R.C.H., Dearle, A., Morrison, R. & Brown, A.L. "Existentially Quantified Types as a Database Viewing Mechanism". *Advances in Database Technology - EDBT90*, Venice. In **Lecture Notes in Computer Science**. 416. Springer-Verlag (1990), pp. 301-315.

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

[DH66] Dennis, J.B. & Van Horn, E.C. "Programming Semantics for Multiprogrammed Computations". *Comm. A.C.M.*, 9, 3, (1966), pp 143-145.

[Fab74] Fabry, R.S. "Capability Based Addressing". *COMM.ACM*, 17,7, (1974), pp. 403-412.

[Feu73] Feustal, E.A. "On the Advantages of Tagged Architecture". *IEEE Transactions on Computers*, C-22, 7, (July 1973), pp. 644-656.

[AGO89] Albano, A., Ghelli, G. & Orsini, R. "Types for Databases: The Galileo Experience". *Proc. 2nd International Workshop on Database Programming Languages*, Oregon, (June 1989), pp 196-206.

[HI85] Hsu, T. & Imielinski, T. "Integrity Checking for Multiple Updates". *Proc. of the ACM-SIGMOD International Conference on Management of Data*, (1985), pp. 152-168.

[Lor77] Lorie, R.A. "Physical Integrity in a Large Segmented Database". *ACM Transactions on Database Systems*, 2, 1, (March 1977), pp. 91-104.

[JL78] Jones, A.K. & Liskov, B. "A language extension for expressing constraints on data access". *Comm.ACM* 21, 5 (1978), pp. 358-367.

[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).

[MB80] Myers, G.J. & Buckingham, B.R.S. "A Hardware Implementation of Capability-Based Addressing". *Operating Systems Review*, 14, 4, (1980).

[MBD88] Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P. "Flexible Incremental Binding in a Persistent Object Store". *ACM.Sigplan Notices*, 23, 4 (April 1988), pp. 27-34.

[MH89] McCune, W. & Henschen, L. "Maintaining State Constraints in Relational Databases". *Journal of the ACM* 36, 1, (January 1989), pp. 46-68.

[MP88] Mitchell J.C. & Plotkin G.D. "Abstract Types have Existential type". *ACM TOPLAS* 10,3 (July 1988), pp. 470-502.

[PS87] "The PS-algol Reference Manual fourth edition". Universities of Glasgow and St.Andrews PPRR-12 (1987).

[RA85] Rosenberg, J. & Abramson, D.A. "A Capability-Based Workstation to Support Software Engineering". *Proceedings of 18th Annual Hawaii International Conference on System Sciences*, (1985), pp. 222-230.

[RHB90] Rosneberg, J., Henskens, F.A., Brown, A.L. & Morrison, R. "Stabilitity in a Persistent Store based on Large Virtual Memory". *Proc of the International Workshop on Security and Persistence of Information*, Bremen, West Germany (1990).

[SFS90] Stemple, D., Fegaras, L., Sheard, T. & Socorro, A. "Exceeding the Limits of Polymorphism in Database Programming Languages". *Advances in Database Technology - EDBT90*, Venice. In **Lecture Notes in Computer Science**. 416. Springer-Verlag (1990), pp. 269-285.

[SS89] Sheard, T. & Stemple, D. "Automatic Verification of Database Transaction Safety". *ACM Transactions on Database Systems* 12, 3 (September, 1989), pp. 322-368.

[SWB89] Schmidt, J.W., Wetzel, I., Borgida, A. & Mylopoulos, J. "Database Programming by Formal Refinement of Conceptual Design". *IEEE - Data Engineering*, (September 1989).

[WLH81] Wulf, W.A., Levin, R. & Harbison, S.P. "HYDRA/C.mmp: An Experimental Computer System". McGraw-Hill, New York, (1981).

[WN79] Wilkes, M.V. & Needham, R.M. "The Cambridge CAP Computer and its Operating System". Elsevier North Holland, Inc., (1979).