Subtyping and Assignment in Database Programming Languages

Richard Connor David McNally Ronald Morrison

Department of Mathematical and Computational Sciences
University of St Andrews
North Haugh
St Andrews
Scotland
KY16 9SS.

{richard,djm,ron}@cs.st-andrews.ac.uk

Abstract

Our focus of interest is in the integration of programming languages and database management systems. In particular, the integration of type systems and data models is considered. One tension in this integration occurs when a type system with subtype inheritance is combined with a data model which contains mutable values.

A description of some well-known problems in such systems is given. This is followed by a classification of possible trade-offs between the safety of static checking, normally required by type systems, and the flexibility of dynamic checking, normally found in data models. At each stage in the classification decreasing static safety is traded for an increasing class of correct programs which may be written. The purpose of this classification is to allow a DBPL designer to understand the implications of any particular type system with both subtyping and mutable values.

1 Introduction

Type systems provide two important facilities within programming languages - data modelling and protection [AM85a]. The type system allows the programmer to partition the universe of discourse of the application into well defined sets of entities that may then be manipulated in a consistent manner. The user models the application in terms of these sets of entities which are the value sets of the types. Thus with regard to modelling, the type system provides a facility close to that of a data model in a database management system. Indeed, a major motivation in developing more and more sophisticated type systems for database programming languages is to integrate the separate traditional activities of type systems and data models [AM86].

It is with regard to protection that type systems and data models differ most. Traditionally it is expected that consistent use of a type system may be checked by a static scan of the program text. This has a number of advantages including discovering errors earlier and, by eliminating run time checks, making programs execute more efficiently. There are very few languages, however, that adhere strictly to static type checking and often dynamic type checks at certain points in the computation, such as the input of values and the projection of unions, are used to maintain the otherwise static invariants.

Data models are strongly typed in that the use of the entities in the models must adhere to the rules of the data model. However, the manipulation of the model is usually such that the use of the entities may only be checked dynamically.

Dynamic type checking has been provided for explicitly in the strongly and mostly statically typed languages Amber [Car85], with type **dynamic**, and Napier88 with type **any** [MBC88]. For persistence the device of dynamic type checking has been used to verify the type consistency of the late binding of persistent data to programs [ABC83, ABM88]. This allows all the separately prepared units of persistent data to be statically checked but their composition to be verified dynamically. This mixture of static and dynamic checking has been found to provide a good balance between static safety and dynamic flexibility [AM85b].

When integrating type systems and data models a spectrum of type checking regimes needs to be considered. Different parts of the spectrum yield a different modelling potential as well as a different balance between the safety of static checking as required by type systems and the flexibility of dynamic checking as required by data models. Henceforth in this paper, the traditional notions of type systems and data models will be considered to be part of a larger database type system for checking constraints on the use of data. It is noted that the similarity between a schema and a type in a database programming language has already been observed [AM86].

The integration of type systems and data models in DBPLs is considered in the context of systems that support subtyping and mutable values. There are some well-known problems with this. A spectrum of possible trade-offs between static safety and flexibility, while always preserving strong typing, is proposed. At each stage in this spectrum decreasing static safety is traded for an increasing class of correct programs which may be written. The classification will allow the DBPL designer to understand the type checking implications of a particular type system with subtypes and mutable values.

1.1 Subtyping in DBPLs

Cardelli [Car84] has proposed a semantics for subtyping in which a type is a subtype of another if the operations allowed on the second type are also allowed on the first. This subtype relation defines a partial ordering of the types, which may therefore be described in terms of a lattice. For simplicity only record types will be considered in order to concentrate on the interaction between subtyping and mutable values. It is noted that Cardelli has also given a semantics for subrange, variant and function subtyping.

A record type τ is defined to be a subtype (written $\leq$) of type τ' if τ has all the fields of τ', possibly some more, and that the common fields of τ and τ' are in the $\leq$ relation. These ideas will be introduced by example. The syntax that is introduced is deliberately verbose in an attempt to free the discussion from considerations of type inference and any particular type equivalence and inheritance rules. For example, all the program segments are valid for both name and structural type equivalence, and for explicity and implicit inheritance. The declarations

> **type** address **is** [place : **string**]
> **type** thing **is** [name : **string**]
> **type** person **is** thing **with** [addr : address]

define three record types *address*, *thing* and *person*. The type *address* is a record, denoted by the square brackets, with one field called *place* of type string. The type *thing* is a record with one field called *name* of type string. The type *person* is also a record but has two fields, one from *thing* called *name* of type string, and the other called *addr* of type *address*. The subtyping rule states that *person* is a subtype of *thing*. The type *person* is defined here as an extension of type *thing*, with an added field *addr* of type *address*. The **with** construct also allows field names to be redefined in the extended type by subtypes of the named field in the supertype. To create a value of type *thing* the following might be used.

> **let** lorry : thing := thing [name := "lorry"]

declares the identifier *lorry* of type *thing* to be a record value of type *thing* with one field called *name* with the string value "lorry". The square brackets and the type name delimit the record constructor. The order of the field declarations is not important but all fields must be specified. A person may be similarly created. For example

> **let** jack : person := person [name := "Jack", addr := address [place := "Kilconquhar"]]

The primary intuition behind subtyping, called the principle of substitutability [WZ88], is that a value of a subtype can be substituted anywhere a value of a supertype is expected. Thus a procedure written for *thing*s should work for *person*s. For example

> **let** thingName : thing → **string** := **proc** (it : thing) : **string** ; it.name

declares a procedure called *thingName* which takes as a parameter a value *it* of type *thing* and returns a string which is the *name* field of *it*. The type of the procedure is denoted as thing → **string**. It is expected that this procedure will work equally well for *thing*s and all its subtypes. The calls

```
thingName (lorry)
thingName (jack)
```

are both legal and return the *name* of the respective record values.

The major advantage of subtyping is that it introduces a form of polymorphism, called inclusion polymorphism [CW85], into the language. In the DBPL context inclusion polymorphism can be used for the partial specification of modules and to aid some problems of system evolution [ACO85]. With inclusion polymorphism it is sufficient for a program to specify only the attributes of persistent modules required by that program for the checking to be legal. Other attributes which are of no relevance to the program need not be specified. By utilising this partial specification, systems may be evolved while maintaining the ability of old programs to operate on new data. In certain circumstances this avoids large scale rewriting of systems to accommodate minor changes in data format.

1.2 Substitutability and Identity

As will be seen in the next section it is in the presence of store semantics that the intuition behind the use of subtyping becomes particularly problematical. We contend, however, that a store semantics is essential to DBPLs to allow the sharing of objects. This introduces the notion of identity [KC86] in that two values are not just equal but are the same instance of the value. The concept of identity is central to object-oriented programming which utilises subtyping as its method of constructing inheritance hierarchies.

In database programming languages substitution occurs where actual parameters are substituted for formal parameters in procedure calls, and in assignment statements. Our particular focus of attention in this paper is where substitution preserves identity. The underlying relationship between assignment and identity is that for *a* and *b*, after a := b then *a* and *b* are identical. Similarly for procedure applications, if *ap* is an actual parameter that is substituted for the formal parameter *fp* in procedure *f* then immediately after the call *f (ap)*, *ap* will be identical to *fp*.

The alternative to such store semantics is copy semantics. Both may be required in a programming language but we contend that copy semantics alone will not meet all the requirements of database programming languages.

1.3 Problems with Subtyping

For the moment some problems introduced by subtyping are considered, without any mention of solutions already known. The discussion starts with the most general case by considering a language whose rule for inclusion polymorphism is the principle of substitutability and where the concept of mutability is not reflected in the type system.

Cardelli [Car84] has shown that the intuition behind subtyping starts to go wrong even without store semantics. Consider for example

```
let thingId : thing → thing := proc (it : thing) : thing ; it
thingId (jack)       ! is of type thing
```

In the above a type widening operation has taken place during the dynamic substitution in the procedure call *thingId (jack)*. That is, the type of the result, *thing*, is a wider type or supertype of *person*. Cardelli originally called this a type checking anomaly. It has also been referred to as a lack of closure in the type system [ADG89].

The same effect occurs with assignment. For example

```
let secondJack : thing := jack
```

declares *secondJack* to be of type *thing* and assigns *jack* to it. The substitution is valid but a loss of type information, due to the type widening, has occurred since *secondJack* is of type *thing* and not of type *person*.

The problem of mutable values is more important. Liskov and Jones [JL78], Albano [Alb83] and Wegner and Zdonik [WZ88] have independently discovered in different contexts a more serious loss of type information which may lead to a program failure in a strongly typed language. Consider the example in Figure 1, where an assignment made to a location later causes a dereference operation to fail.

```
type address is [place : string]
type thing is [name : string]
type person is thing with [addr : address]

type car is [model : string ; colour : string]
type employee is person with [employeeNo : int ; vehicle : car]

let changeCar : employee → void := proc (e : employee) : void
          e.vehicle := car [model := "model T Ford", colour := "black"]

type Jaguar is car with [extraFuelTanks : int]
type director is employee with [vehicle : Jaguar]

let ronald : director := director [
          name := "Ronald",
          addr := address [place := "St Andrews"],
          employeeNo := 1,
          vehicle := Jaguar [model := "Etype", colour := "red", extraFuelTanks := 1]]

changeCar (ronald)
let f : int := ronald.vehicle.extraFuelTanks      ! will cause an error since the
                                                   ! extraFuelTanks field no longer
exists
```

Figure 1 Problems with subtyping

In Figure 1 the type *Jaguar* is a subtype of *car* and the inclusion of *thing* and *person* is extended by *director* and *employee*. The procedure *changeCar* takes an *employee* as a parameter and updates the *vehicle* field with a value of type *car*. The call *changeCar (ronald)* is legal since *director* is a subtype of *employee*. The effect of calling the procedure is to update the *vehicle* field of *ronald*. After the call the program has lost the ability to execute the field selection *ronald.vehicle.extraFuelTanks* correctly since the *extraFuelTanks* field of the record no longer exists. Statically all the individual actions appear to be type safe; however, the program is clearly in some sense incorrect.

The problem just described occurs because the application of the principle of substitutability may not be statically checked for correctness in the language of the example. This is because it is possible to create many different type views over the same updatable location. Some of these views reflect less attributes of the location than others. If an update is specified from a less general view then the new value may have less attributes than expected from a different viewpoint. Thus the update to *e.vehicle* looks sensible within the context of procedure *changeCar*. However it is actually a violation of the principle of substitutability as it substitutes one value with a value of one of its supertypes. That is, *ronald.vehicle* is updated with a value of type *car* which is a supertype of *Jaguar*.

The program in Figure 1 is clearly in error. The possibilities are that any one of the assignment, the field selection, or the call of the procedure is incorrect. The parameter substitution on the call would appear to be correct since a subtype has been substituted for a supertype. It would at first appear that the assignment causes the problem. However the program does not go wrong until the record field selection *ronald.vehicle.extraFuelTanks* is executed.

If the field selection were not executed then the program may terminate correctly, which may lead to the conclusion that it is the selection which is in error. Figure 2 illustrates that in such a system values of unrelated types may be assigned to one another without error, thus defeating the purposes of the type system.

```
type empty is []
type emptyContainer is [it : empty]

let itSwap : emptyContainer,emptyContainer → void :=
        proc (A, B : emptyContainer ) : void
        begin
                let temp : empty := A.it
                A.it := B.it
                B.it := temp
        end

type directorContainer is emptyContainer with [it : director]
type JaguarContainer is emptyContainer with [it : Jaguar]

let thisDirector : directorContainer := directorContainer [it := ronald]
let thisJaguar : JaguarContainer :=
        JaguarContainer [it := Jaguar [model := "Etype",
                                    colour := "red", extraFuelTanks := 1]]

itSwap (thisDirector, thisJaguar)
thisDirector.it.vehicle
```

Figure 2 Swapping types

At the end of the execution of the program segment in Figure 2 *thisDirector.it* has a value of type *Jaguar* and *thisJaguar.it* has a value of type *director*. Again the static typing appears correct, but unrelated types have been assigned to one another and all static type information has been lost. This problem is compounded in languages with a persistent store since over time the user can have very little confidence that the static type information associated with values in the store is in any way accurate.

It should also be pointed out that the problem is nothing to do with either scoping or procedure call, as the example of Figure 3 shows. As the type of *anEmployee* is *employee*, although the value is really of type *director* the assignment to the *vehicle* field appears statically to be correct.

```
let anEmployee : employee := employee [
          name := "Richard",
          addr := address [place := "Tentsmuir"],
          employeeNo := 2,
          vehicle := car [model := "model T Ford", colour := "green"]]

let ronald : director := director [
          name := "Ronald",
          addr := address [place := "St Andrews"],
          employeeNo := 1,
          vehicle := Jaguar [model := "Etype", colour := "red", extraFuelTanks := 1]]

anEmployee := ronald
anEmployee.vehicle := car [model := "model T Ford", colour := "blue"]]

let f : int := ronald.vehicle.extraFuelTanks      ! will cause an error since the
                                                  ! extraFuelTanks field no longer
exists
```

Figure 3 No scoping or procedure calls

In the worst case using such a system, any two values whose types share an upper bound in the type lattice may be viewed as each other. In general a system may retain no meaningful static type information unless some restriction is made on either the subtyping rules or the assignment rules.

2 Categorising the solutions

The problem described in Section 1 is manifested as a conflict between two apparently independent intuitions. The first of these is type accuracy; that is, any static type description associated with a value is an accurate description of that value's attributes. This implies that the operations allowed in a computation must be restricted to guarantee the static assertions of type, but does not necessarily imply totally static type checking. Type accuracy is often referred to as soundness in the context of static type systems. The second intuition, the principle of substitutability, always allows subtype values to be used in substitution operations where supertype values are specified. As will be seen these intuitions are in fact interdependent and must be considered together.

In addition to the assumption that substitution preserves identity, three other assumptions have been made about the language used for the examples in the previous section, and the combination of these was shown to break the conditions of type accuracy. The assumptions were:

- the principle of substitutability itself: a value with more "functionality" may be always used in place of one with less,
- that the description of a value's functionality, i.e. its type, does not necessarily include information about whether its components are mutable or not, and
- that an assignment operation may always be statically determined to be safe.

Each assumption is derived from qualities that may reasonably be desired, the second and third from a language which features mutability and the first from one which features subtyping. It has been shown above that these desiderata are not mutually tenable. It will be shown, however, that type accuracy may be enforced whilst retaining any two of the three, so long as sufficient a restriction is placed on the remaining one.

Zdonik and Maier [ZM90] in a similar treatment of object-oriented languages list similar mutually untenable conditions. They describe one further condition, however, the presence of predicate subtyping, and furthermore state that a system with any three of their four conditions is safe. This difference is attributable to some subtle assumptions made about the object-oriented model, notably that assignment to fields within a value is performed only by an encapsulated procedure declared at the same scope level as the data.

To maintain type accuracy, the system must prevent any dangerous update from occurring. Modifications which are sufficient to preserve the overall type accuracy of the system may be made in any of the following categories:

- substitution context limitation:

  limit the contexts in which substitution using inclusion may occur,
- substitution mutability limitation:

  model mutability within the type system, and restrict type inclusion in some appropriate manner, or
- substitution dynamic failure:

  check the validity of substitution dynamically, and accept that a failure may occur at the time of update.

The danger of making such restrictions is that they may appear arbitrary to a programmer, and therefore complicate the task of understanding a programming language. However, restrictions in all of the above categories have been made for other reasons within various languages and systems. Examples in each category are:

- bounded universal quantification is a mechanism which allows substitution by inclusion only in certain well-defined contexts,
- the type system of ML (and many other languages) models mutability, in order to delimit referential transparency within programs, and

- many database systems impose dynamically checked integrity constraints, which may cause a system to fail as a result of update.

A description of a number of restrictions, each of which preserves type accuracy, is now given. All possible mechanisms are not described but all of the serious contenders for use in a database programming language are believed to be included. They are introduced starting with the system with most static type information and ending with the least. As would be expected, the flexibility of each system increases, and static safety decreases, as less static constraints are imposed. That is, where the ability to express more type correct computations increases the possibility of static detection of errors decreases.

In order to justify the correctness of each proposed system, a little formalism is first introduced to allow an accurate description of the restrictions imposed in each case.

2.1 A little formalism

As the problem described is restricted to values with mutable components, the denotational semantic model of locations as described by Strachey [Str67] is adopted. For each location, notation is introduced to describe a number of associated types.

The first type of interest is the type with which the location is originally created. In strongly typed languages every value is created in a type environment which statically asserts a type for any new location, and this is known as the creation type. For example, after the declaration:

let jack : person := person [name := "Jack", addr := address [place := "Kilconquhar"]]

the creation type of *jack* is *person*, and the creation type of the location denoted by *jack.addr* is *address*.

For any location denoted by *i*, written loc (i), the type attributed to it at the time of its creation will be denoted as

$T_{\mathbf{creation}}$ (loc (i))

Thus $T_{\mathbf{creation}}$ (loc (jack)) is *person*.

Also of interest is the minimum type within the lattice of the value referred to by a location. In a system with inclusion polymorphism, where values may have more than one type, the interest is in the most specific type which accurately describes the attributes of the value. For example, after the declaration

let secondJack : thing := jack

then $T_{\mathbf{creation}}$ (loc (secondJack)) is *thing*, and the r-value minimum type of *secondJack* is *person*. The r-value minimum type of a location varies dynamically. In this case it may be statically determined only because the value *jack* is manifest and not because of its type.

For any location *i*, the r-value minimum type will be denoted as

$T_{\mathbf{r\text{-}value}}$ (loc (i))

Thus in the above $T_{\mathbf{r\text{-}value}}$ (loc (secondJack)) is *person*.

During the execution of a program, a number of different views may be formed over a location. This is the case wherever it is possible to denote the same location by a number of different access paths; that is, when an alias for a location is created. In a system with subtyping, these views may be of different types, thus forming a set of view types with which each location is viewed at any one time. For example, during the execution of the *changeCar* procedure in Figure 1, the location denoted by *e.vehicle* for that particular invocation has the view set

{car, Jaguar}

For any location *i* and view *j*, the type of the view will be denoted as

$T_{\mathbf{view\ (j)}}$ (loc (i))

Since there may be more than one view of the location, the views form a view set for location *i*. For *n* views, this is denoted by

$\{T_{\mathbf{view\ (j)}}$ (loc (i)), j $\leftarrow$ 1..n$\}$

From the context of its creation each location has an associated type lattice which describe the possible sets of its r-value and view types, as shown in Figure 4.

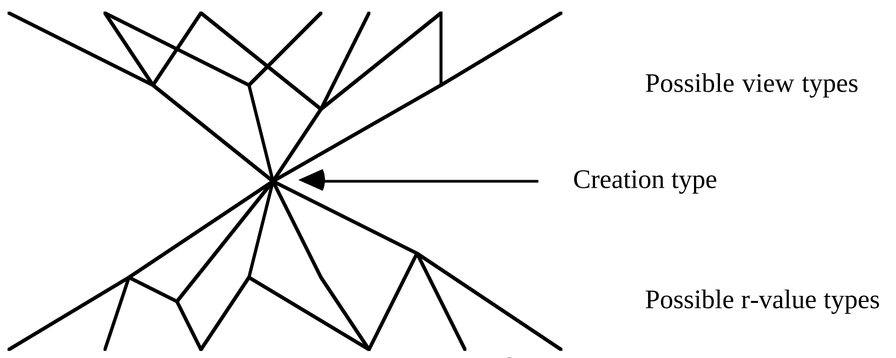

Figure 4 A type lattice for each location

The most general condition for the preservation of type accuracy within a system is that, for any location, the r-value minimum type is a subtype of all of the type views. This is defined as the type accuracy invariant:

$$\forall_i . \forall_j . T_{\textbf{r-value}} (\text{loc } (i)) \le T_{\textbf{view (j)}} (\text{loc } (i))$$

This invariant may be used as a post-condition for any substitution operation to preserve type accuracy. As will be seen later in certain circumstances the preservation of this invariant can be checked statically while in others it must be checked dynamically. In either case the static type descriptions are accurate.

For any location, the evaluation of a substitution operation may affect either the r-value minimum type or the set of view types, according to whether the location is on the left or right hand side of the substitution operator. The intuition behind the principle of substitutability is that the r-value minimum type may be any subtype of the creation type, and that every member of the set of view types may be any supertype of the creation type. This may be represented by the following inequality:

$$\forall_i . \forall_j . T_{\textbf{r-value}} (\text{loc } (i)) \le T_{\textbf{creation}} (\text{loc } (i)) \le T_{\textbf{view (j)}} (\text{loc } (i))$$

Again this invariant may be used as a post-condition for substitution operation. It is of interest in that it implies the conditions for type accuracy within a system.

The type accuracy invariant was not preserved in the erroneous examples shown in Figures 1 and 2. In both cases this was due to an assignment to a location after which the r-value minimum type was no longer a subtype of all the view types. If, in Figure 1, the location of the *vehicle* field of the *director* object is denoted by *X* then $T_{\textbf{creation}}$ (loc (X)) is *Jaguar*. $T_{\textbf{view (ronald.vehicle)}}$ (loc (X)) is also of type *Jaguar* and $T_{\textbf{view (e.vehicle)}}$ (loc (X)) is of type *car*. After the assignment $T_{\textbf{r-value}}$ (loc (X)) is also of type *car* and is now a supertype of one of the views. Thus the substitution violated the invariant.

$$\forall_i . \forall_j . T_{\textbf{r-value}} (\text{loc } (X)) \le T_{\textbf{view (ronald.vehicle)}} (\text{loc } (X))$$

This was not statically detected as the assignment was specified in the context of one of the view types. It may be seen that the principle of substitutability itself is not incorrect; merely the intuition that the correctness of its application is statically checkable.

In the rest of the paper, the discussion is of how the three categories of restriction mentioned earlier may be imposed within a programming system so that type accuracy is always preserved. In each case the formalism will be used to give a justification that the method preserves the type accuracy invariant.

2.2 Discussion of mechanisms

The following three sections include discussion of six different mechanisms by which type safety may be guaranteed. These fall into the three categories given in Section 2. For each mechanism, the following treatment is given:

1. A description of the language model
2. How the failure shown in Figure 1 is captured
3. Use of formalism to show how the general invariant is preserved

4. A brief discussion of the model, in terms of expressibility and static safety

The following example from a database schema is used throughout.

Consider a database which models company employees. Each employee is allocated a car from the company pool, and this allocation may be changed from time to time. Company directors, however, are always given Jaguars. Jaguars have more attributes than ordinary cars. This is described by the schema fragment in Figure 5.

```
type address is [place : string]
type thing is [name : string]
type person is thing with [addr : address]
type car is [model : string ; colour : string]
type Jaguar is car with [extraFuelTanks : int]
type employee is person with [employeeNo : int ; vehicle : car]
type director is employee with [vehicle : Jaguar]
```

Figure 5 The company schema

The following have been identified as desirable features to model within such a database system:

1. a collection of all Jaguars
2. a collection of all directors
3. a collection of all cars (including Jaguars)
4 a collection of all employees (including directors) who will never change their cars
5. a collection of all employees (including directors)
6. a procedure which gives any employee's number
7. a procedure which swaps the vehicles of either two directors or two non-directors
8. a procedure which allocates a new car to a non-director (using the collection of all employees)
9. a procedure which allocates a suitable new car to any employee (including directors)
10. a procedure which demotes a director, and allocates a new vehicle which is not a Jaguar
11. a procedure which allocates a Jaguar to an employee who is then made a director

3 Substitution context limitation

This section describes in detail a single mechanism where the context in which subtype inclusion is allowed is restricted to preserve type accuracy. The mechanism shown restricts inclusion to only those places where it is explicitly denoted by the use of bounded universal quantification. In any other context type descriptions are exact. Bounded universal quantification may be fully statically checked and works by preventing the creation of supertype views over any component values.

There are other possible mechanisms which achieve type accuracy by a restriction of where subtyping may be employed, such as a rule which allows subtyping only over locations which may never be aliased. Such a system is employed in the record subtyping of Modula-3 [CDG88], where it is achieved by the combination of a non-recursive subtype relation and the implicit evaluation of locations to r-values.

Also of interest in this context is a type system described by Ohori et al [BO91], which achieves the power of subtyping by the introduction of bounded union types into an exactly typed system. Values may be explicitly injected into and projected from appropriate union types, with the bounds on the unions describing some minimum properties of their behaviour.

Although the system is described in the absence of update, its exact typing is a sufficient restriction to preserve the accuracy of static type restrictions.

3.1 Bounded Universal Quantification

Bounded universal quantification is considered in isolation for the sake of clarity, although it may normally be expected to be used in conjunction with another form of subtyping or along with bounded existential quantification. The exclusive use of bounded universal quantification as a subtyping mechanism restricts subtyping to those cases where an explicit type quantifier is declared to stand for a type with at least a given set of properties. Quantified procedures may be written to operate over values with the desired properties and the correct application of these procedures may be statically determined from their calls. As all type quantification is explicit, values of one type may never be viewed as a different type. Bounded universal quantification may also be used in conjunction with any of the other forms of subtyping described in this paper.

The notation used here to describe bounded universal quantification is adapted from [CW85], and is introduced by way of an example:

```
type t is ...

let a := proc [s ≤ t] (x, y : s) : s ; ...
```

This code describes a procedure *a* which takes two arguments *x* and *y* of the same type, and returns a result also of the same type. This type is explicitly abstracted over and is denoted by *s* within the procedure. The type represented by the quantifier *s* is bounded by the type named as *t*; that is, the type *s* is a subtype of *t*. Thus any values of type *s* have at least all of the properties associated with type *t*, although they are not type compatible with values of type *t*.

Bounded universal quantification is often seen as a mechanism by which information loss may be avoided within a more general subtyping system. For example, instead of

```
let thingId : thing → thing := proc (it : thing) : thing ; it
```

which caused a type information loss problem when applied to any proper subtype of *thing*, an equivalent quantified procedure could be written:

```
let thingId := proc [t ≤ thing] (it : t) : t ; it
```

This procedure may be called with parameters of different types, for example

```
let x := thingId [person] (jack)
let y := thingId [director] (ronald)
```

and the type of the result is the same as the argument. This is because the use of the identifier *t* within the procedure description statically asserts the parameter and result type to be the same type, which may be any subtype of *thing*. At a call of this procedure, the type of the result may therefore be deduced as the same as that of the parameter, although this type may be different for each invocation of the procedure. The type of the procedure body must statically be deduced not as any subtype of *thing*, but in particular the same type as the parameter *it*.

Implicit quantifiers may appear in a system with only bounded universal quantification. For example, consider the example in Figure 6.

```
let x := proc [t ≤ employee] (p : t)
begin
        . . .
        let CAR := p.vehicle
        . . .
end

x [person] (jack)
x [director] (ronald)
```

Figure 6 An implicit quantifier

Within the body of the procedure, the type of *CAR* is not known. All that may be determined statically is that the type of *CAR* is some subtype of *car*. This is not type compatible with any other value except those derived from a dereference of a value of type *t*. The type of *CAR* is some implicit quantifier type such as "*s* such that *s* ≤ *car*".

In a type system where the subtype rule describes a single lattice, and therefore any two types have a common lower bound, then bounded quantification alone may have limited expressive power. Such systems typically have a single value of type ∀t.t whose type is a subtype of any other type. In such a system, the procedure

let thingName := **proc** [t ≤ thing] (it : t) : **string** ; it.name

is not type correct. This is because the value of *it.name* is of an implicit quantifier type "*s* such that *s* ≤ **string**", and there is no static information to show that *it.name* is not of type ∀t.t rather than **string**.

There is no obligation for the subtype relation to describe a single lattice rather than a number of discrete ones, where a separate lattice describes subtyping over each type constructor over which the subtype relation is defined. This allows more bounded universally quantified procedures to be written. If subtyping is defined only over record types, for instance, then the type of any implicit quantifier which is not a record type may be deduced. The implicit quantifier "*s* such that *s* ≤ **string**" would describe only the type **string** as this type has no other subtypes. In such a system the above *thingName* procedure is type correct.

3.1.1 Failure capture

It can be seen that an attempt to write the procedure *changeCar* whose call resulted in error before will fail in a system with only bounded universal quantification:

let changeCar := **proc** [t ≤ employee] (e : t) : **void**
e.vehicle := car [model := "model T Ford", colour := "black"]

This is because the type of *e.vehicle* is an implicit quantifier type, "*s* such that *s* ≤ *car*", and as such is not type compatible with a value of type *car*. Thus the procedure is not type correct.

To allow a procedure which assigns correctly to such a location the ability to declare related quantifiers is required, so that the implicit quantifier may be made explicit. For example

let changeCar := **proc** [s ≤ car, t ≤ [vehicle : s]] (emp : t ; Car : s) ; emp.vehicle := Car

describes a procedure with two related quantifiers. The first, *s*, is stated to be a subtype of *car*, and the second, *t*, is stated to be a subtype of any record type which includes a field *vehicle* of type *s*. In this way it is determined statically that the assignment is correct, as otherwise a call of the procedure would fail. For example, the call

changeCar [Jaguar, director] (ronald, newJaguar)

is allowed, but

changeCar [car, director] (ronald, newFord)

is disallowed statically when the type of the call is checked, as values of type *director* do not have a *vehicle* field of type *car*.

Relationships between quantifiers may also be used to further refine the type of procedure arguments in other ways. Consider for example the following procedure:

let breed := **proc** [t ≤ animal,
t1 ≤ t **with** [gender : male],
t2 ≤ t **with** [gender : female]] (father : t1 ; mother : t2) : t ...

of which the call

breed [dog, maleDog, femaleDog] (Gnasher,Fifi)

will produce a dog (Gnipper?) if you have read the Beano [DCT91]. The subject of bounded quantifier relationships in general is an interesting research topic but outside the scope of this paper, and is discussed further in [CM91].

3.1.2 Formalism

Allowing only bounded universal quantification preserves the system invariant described as it prevents any supertype viewing over locations. As there is no way of constructing different type views of a location, all views of the same location from different contexts are the same type:

$$\forall_i.\forall_j.T_{\textbf{creation}}\ (\text{loc}\ (i)) = T_{\textbf{view (j)}}\ (\text{loc}\ (i))$$

Neither is it possible to assign a subtype to a location, and so all r-values have the same type as their locations:

$$\forall_i . T_{\textbf{r-value}} (\text{loc } (i)) = T_{\textbf{creation}} (\text{loc } (i))$$

Together these give the following, which trivially satisfies the type accuracy invariant:

$$\forall_i . \forall_j . T_{\textbf{r-value}} (\text{loc } (i)) = T_{\textbf{view (j)}} (\text{loc } (i))$$

3.1.3 Modelling ability

The modelling power of bounded universal quantification alone is less than other subtyping mechanisms. In the employee/vehicle database, it is unable to model collections of all cars or all employees. However, it is able to model the simpler procedures which require subtyping, and it does not require mutability to be reflected in the type system.

There exists a large class of procedures however where related quantifiers exist, such as a procedure which will supply any employee with an appropriate car, where both the employee and the car are passed as parameters. Bounded universal quantification alone allows the accurate description of such procedures. As bounded universal quantification is generally orthogonal to other forms of subtyping it may be used in conjunction with another system to allow this class of procedure.

4 Substitution mutability limitation

The second category of solution to ensure type accuracy depends upon restricting the subtyping relation. Such restrictions prevent update to locations which are viewed over. To achieve this it is necessary for mutability to be modelled within the type system. Some new syntax is first introduced to denote mutability within types.

The discussion of mutability will be restricted to locations which are contained within other data objects, such as fields of a record. In previous examples, record type definitions have included an implicit assumption that all locations described are mutable; now structure fields may be declared as either updatable locations or constant values. The reserved words **loc** and **val** are now included in the type descriptors to indicate this. Thus for example,

type address **is** [place : **val string**]
type person **is** thing **with** [addr : **loc** address]

describes two types *address* and *person* as before, but now the *place* field of *address* is declared to be a non-mutable value, whereas the *addr* field of *person* is declared to be a mutable location.

Mutability is modelled in a number of languages which do not include any concept of subtyping. For example, the language ML [Mil83] is mainly applicative in flavour, but also includes mutable locations. The mutability of such locations is explicit within the type system. In ML it is possible to preserve referential transparency associated with purely applicative languages for program segments which do not use any mutable types.

The two mechanisms described in this section are already in use in languages with subtyping. Galileo [ACO85] disallows subtyping over any mutable values, in the way described in section 4.1, and Quest [Car89] allows a mutable location to be modelled as a subtype of a non-mutable value as described in section 4.2.

4.1 No subtyping over locations

A first approach to restricting the subtyping rule in a way which preserves type safety is to allow subtyping only over values which are not mutable. Only trivial subtyping is allowed over locations, expressed by the rule:

loc T ≤ **loc** S iff T ≤ S and S ≤ T

That is, mutable locations of types *S* and *T* are type compatible if and only if *S* and *T* are the same type. Such a rule is used in the type system of the language Galileo.

4.1.1 Failure capture

Any potentially dangerous update to a location is detectable and disallowed by a static scan of the program in this system. The program fragment

```
type employee is person with [employeeNo : val int ; vehicle : loc car]
type director is employee with [vehicle : loc Jaguar]

let changeCar : val employee → void := proc (e : val employee) : void
                    e.vehicle := car [model := "model T Ford", colour := "black"]
```

is allowed, but the call

```
changeCar (ronald)
```

is not allowed, as *director* is not defined as a subtype of *employee* due to its variable location *vehicle*.

4.1.2 Formalism

As there is no subtyping over mutable locations in this system, static typechecking ensures that no supertype views are created over such locations:

$$\forall_i . \forall_j . T_{\textbf{view (j)}} (\text{loc } (i)) = T_{\textbf{creation}} (\text{loc } (i))$$

This implies that all views of a location have the same type:

$$\forall_i . \forall_j . \forall_k . T_{\textbf{view (j)}} (\text{loc } (i)) = T_{\textbf{view (k)}} (\text{loc } (i))$$

For any view of a location, the subtyping rules do allow any subtype of a particular view type to be assigned. After an assignment to any location *i* from any context *j*, the following holds:

$$T_{\textbf{r-value}} (\text{loc } (i)) \leq T_{\textbf{view (j)}} (\text{loc } (i))$$

which, as all view types are the same, gives the type accuracy invariant:

$$\forall_i . \forall_j . T_{\textbf{r-value}} (\text{loc } (i)) \leq T_{\textbf{view (j)}} (\text{loc } (i))$$

4.1.3 Modelling ability

This system is able to model heterogeneous collections, such as the collection of all cars including Jaguars. It cannot, however, model the collection of all employees including directors, because the *vehicle* field of employee is mutable and is a supertype of the *vehicle* field of director, causing the two types to be unrelated in the subtype lattice.

4.1.4 Miscellaneous

The inability to create even non-updatable views over locations seems over-restrictive in terms of polymorphism. For example, if two types are declared as

```
type thing1 is [name : val string]
type thing2 is [name : loc string]
```

then it is not possible to write a single procedure which will extract the *name* field of values of either type, as, for example

```
let nameOf : val thing1 → string := proc (x : val thing1) : string ; x.name

nameOf (thing2 [name := "peter"])
```

is not type correct. The next section shows a system which allows such polymorphism, although some further problems are introduced.

4.2 Only non-mutable views allowed

The above system may be extended without losing its statically determinable safety by allowing a mutable value to be a subtype of a non-mutable value but no other subtyping over mutable values. Thus the rule for subtyping over locations still stands:

loc T ≤ **loc** S iff T ≤ S and S ≤ T

but a new rule is also introduced, which states

loc T ≤ T

for any type T.

In practical terms this means that a location may be viewed as any of its supertypes, as long as the ability to update this location is relinquished. Where the location is of an updatable type it is known statically that this type is the same as the type with which the location was created.

Thus an update to a location may only occur using a value which is a subtype of the creation type. Such a rule occurs in the type system of the language Quest.

4.2.1 Failure capture

Failure capture in this system is identical to that in the previous section. Once again, the program fragment

```
type employee is person with [employeeNo : val int ; vehicle : loc car]
type director is employee with [vehicle : loc Jaguar]

let changeCar : val employee → void := proc (e : val employee) : void
                        e.vehicle := car [model := "model T Ford", colour := "black"]
```

is allowed, but the call

```
changeCar (ronald)
```

is not allowed, as *director* is not defined as a subtype of *employee* due to its variable location of type *car*. If type *employee* is specified as having a non-mutable *vehicle* field then the call would be allowed as **loc** *Jaguar* is a subtype of **val** *car*. In this case the procedure itself would of course be disallowed as it now specifies an update to a location which is not specified as mutable.

4.2.2 Formalism

In this system it is possible to create views with different types over a mutable location. In the absence of assignment, the static typechecking ensures that all views created are supertypes of the creation type:

$$\forall_i.\forall_j.T_{\textbf{creation}}\ (\text{loc}\ (i)) \leq T_{\textbf{view (j)}}\ (\text{loc}\ (i))$$

However, views may only be created when the ability to update a location is relinquished and so updates may only occur with subtypes of the location's creation type:

$$\forall_i.T_{\textbf{r-value}}\ (\text{loc}\ (i)) \leq T_{\textbf{creation}}\ (\text{loc}\ (i))$$

which, in conjunction with the previous inequality, gives the required invariant:

$$\forall_i.\forall_j.T_{\textbf{r-value}}\ (\text{loc}\ (i)) \leq T_{\textbf{view (j)}}\ (\text{loc}\ (i))$$

4.2.3 Modelling ability

The restrictions of introducing mutability into the type system are clearly lessened by this additional subtyping rule since mutable values may be viewed as non-mutable. This means that the impact of type incompatibility between such values is lessened; for example, the *thingName* function above may be applied to values with either mutable or non-mutable *name* fields, as the function does not attempt to update the field and so may view it as a non-mutable value. In terms of data modelling, a collection of values some of which have mutable components and some of which do not may be formed, but the collection must be of the type with the non-mutable components.

As an example from the employee/vehicle database, it is possible to form a collection to model all company employees including directors; however, this may only be achieved by relinquishing the right to update the vehicle field of an employee from the context of this collection.

4.2.4 Miscellaneous

One point that should be mentioned is that the argument for including mutability in the type system to allow the demarcation of referential transparency has now been lost. This occurs where a value which is viewed as non-mutable is updated from a different context. Consider the example in Figure 7.

```
type X is [x : val int]
type Y is [x : loc int]

let Y1 : Y := Y [x := 3]

let changeConstant : val X → bool := proc (a : val X) : bool
                      begin
                           let A := a.x
                           Y1.x := 4
                           A = a.x
                      end

let Z = changeConstant (Y1)  ! evaluates to false
```

Figure 7 Changing apparently non-mutable values

In Figure 7, the value *a.x* within procedure *changeConstant* is regarded as non-mutable. However by using its global identifier it can be updated by another view. Mutability is now only modelled by the type system so far as "known to be mutable" or "no information about mutability" for each component value.

The safety of static assignment checking will appeal in particular to a programming language designer. A database system designer, however, will be less worried about checking such constraints dynamically, as many other integrity constraints in databases are not statically checkable, but may be more concerned about the associated loss of flexibility.

5 Dynamic Assignment Checking

The final category of solutions contains those in which the preservation of type accuracy is ensured by dynamically checking the validity of updates. The models in this category allow more expressibility than those in the statically checkable categories. They have two main drawbacks: it is necessary to define failure semantics for assignment and the execution cost of the dynamic check may be significant. Three mechanisms in this category are shown with varying degrees of expressibility, static safety and execution cost.

Database management systems traditionally rely more upon dynamic constraint checking than programming languages. Such constraints are often finer-grained than static type constraints and in some sense are intended to govern the acceptable dynamic behaviour of the system. The ability to check all such constraints statically is beyond the ability of current type checkers and theorem provers.

The parallel between a traditional database integrity model and a programming language type system may be extended to encompass the problems of subtyping and update. For example, the subtype relation between employees and directors would be partly modelled in a traditional database by an integrity constraint which states that an employee who is also a director must have a vehicle which is also a Jaguar. This constraint would normally be implemented as a precondition check on an assignment to the *vehicle* field of such a value. In such a system the erroneous program in Figure 1 would be halted at the assignment due to the violation of this constraint.

In a well defined data model, it may be that the integrity constraints are sufficiently fine-grained to prevent the execution of any program which causes a violation of type accuracy. In this case, no further type system restriction is necessary as the execution of any operation which violates type accuracy will cause the program to be terminated. However, except in those cases where the preservation of type accuracy may be guaranteed by an analysis of the set of integrity constraints, a dynamic constraint which enforces type accuracy is also necessary.

The most straightforward model of dynamic checking is a check that the right hand side of an assignment statement is a subtype of the creation type of the location being assigned to. This corresponds to a dynamic check of the correct application of the principle of substitutability but may be expensive to implement. This method is shown in section 5.2. However, there are two other useful possibilities, which are also described here. Section 5.1 describes a view equality check which is more restrictive but allows a more efficient dynamic check. The other mechanism, described in section 5.3, is a dynamic view set check, which allows the maximum possible flexibility that maintains type accuracy within a system.

With the introduction of dynamic checking on assignment, the static typechecking rules for assignment must also be reconsidered. Systems which are entirely statically checked necessarily include a rule that on an assignment

<e1> := <e2>

where *<e1>* and *<e2>* denote locations *i* and *j* respectively, then

$T_{\mathbf{view\ (e2)}}\ (\mathrm{loc}\ (j)) \leq T_{\mathbf{view\ (e1)}}\ (\mathrm{loc}\ (i))$

Although necessary to preserve type accuracy in the static systems shown, a dynamic check on assignment precludes the necessity for this static restriction. Nevertheless a programming language designer may wish to retain it in conjunction with a dynamically checked system as it increases the static safety of a program. Once again, however, greater flexibility may be achieved if the static restriction is removed. The systems described in Sections 5.1 to 5.3 assume that this static constraint is imposed and systems with no static constraints on assignment are considered in Section 5.4.

5.1 View equality check

This mechanism is similar to that described in Section 4.2, in that update is allowed only from views of locations which have the same type as the creation type. The difference is that with this mechanism mutability is not modelled within the type system. This allows a more flexible data model but in general it is not possible to determine statically every place where a dangerous update may occur.

When an assignment is specified in such a system there is an implicit assumption that the type view of the location being assigned to is the same as the creation type of the location. Thus in the procedure

```
let changeCar : employee → void := proc (e : employee) : void
                              e.vehicle := car [model := "model T Ford", colour := "black"]
```

the assignment operation implicitly includes an assertion that the creation type of the location denoted by *e.vehicle* is precisely *car*. The assertion is made from the fact that *e* is defined as type *employee* and that the *vehicle* field of *employee* is of type *car*. The static type system only restricts the actual parameter value substituted for *e.vehicle* to have a creation type which could be any subtype of *car*. A dynamic test is performed before the assignment and if the creation type of the location within the value denoted by *e* is not of precisely *car* then the program will fail at this point.

It should be noticed that the right hand side of the assignment statement could be any subtype of *car*. This use of subtyping may be checked statically.

5.1.1 Failure capture

Some incorrect programs can only be detected dynamically in this system. In the *changeCar* example, the procedure is statically allowable. It is also statically allowable to call the procedure with an actual parameter of any subtype of *employee*. Any such value contains a mutable location *vehicle* which may be any subtype of *car*. However, the assignment statement carries the assertion that *e.vehicle*, when evaluated, will provide a location with a creation type of precisely *car*. This will cause a dynamic failure when the procedure is called with the value *ronald* of type *director*, whose *vehicle* field has a creation type of *Jaguar*.

Recall that in Figure 1 the test could be performed at the procedure call rather than at the assignment. This is not in general the case since the location *e* is type compatible with any other value of a subtype of *employee* and could have been assigned a different value before the assignment to the *vehicle* field is executed.

5.1.2 Formalism

In formal terms, this dynamic check ensures that whenever an update occurs to a location *i* from a view *j* then

$T_{\mathbf{view\ (j)}}\ (\mathrm{loc}\ (i)) = T_{\mathbf{creation}}\ (\mathrm{loc}\ (i))$

The static typechecking performed by the system simultaneously ensures that after any assignment from the same view *j*

$T_{\mathbf{r\text{-}value}}\ (\mathrm{loc}\ (i)) \leq T_{\mathbf{view\ (j)}}\ (\mathrm{loc}\ (i))$

and so, after any assignment to a location i

$T_{\textbf{r-value}} (loc\ (i)) \leq T_{\textbf{creation}} (loc\ (i))$

The static subtyping rules also ensure that only supertype views of a location may be constructed:

$\forall_i . \forall_j . T_{\textbf{creation}} (loc\ (i)) \leq T_{\textbf{view (j)}} (loc\ (i))$

The combination of these inequalities gives the required invariant in the presence of assignment:

$\forall_i . \forall_j . T_{\textbf{r-value}} (loc\ (i)) \leq T_{\textbf{view (j)}} (loc\ (i))$

as required.

5.1.3 Modelling ability

The modelling power of this system is greater than any of the purely static systems. In terms of the example requirements over the employee/vehicle database the view equality test allows the modelling of a collection of all employees. This is not possible in any of the statically checkable systems since the type of a director's vehicle is a subtype of the other employees' vehicles. Once this collection has been established an employee who is not a director may have the *vehicle* field updated, but any attempt to update a director's vehicle when the director is viewed as a member of this collection will fail as the view type is different from the creation type. Notice that this update will fail even if the value assigned is a subtype of *Jaguar*.

5.1.4 Miscellaneous

The dynamic model described here is slightly more restrictive than other dynamic models. An advantage of this dynamic test, however, is that it may be relatively efficient. This is because in general it requires a type equivalence test rather than a subtype test. In many cases this equivalence test may be highly optimised [CBC90]. This is very much in contrast with the more general dynamic test described in the next section where potentially a full structural subtype test may be necessary.

5.2 General subtype check

This technique is a mechanism which implements the principle of substitutability without any restriction. That is, an instance of a subtype may be substituted anywhere that an instance of a supertype is expected. Otherwise a dynamic assignment failure occurs. In terms of our nomenclature this means that an assignment will always succeed whenever a location's creation type is a supertype of the value being assigned. This implies that a full subtype check may be necessary whenever an assignment is executed.

5.2.1 Failure capture

Once again the *changeCar* procedure is statically allowable as is the call with a value of type *director*. The procedure

```
let changeCar : employee → void := proc (e : employee) : void
                        e.vehicle := car [model := "model T Ford", colour := "black"]
```

here carries an implicit assumption that the creation type of the location *e.vehicle* is any supertype of the type *car*. Unlike the system described in 5.1 the implicit type assertion associated with the assignment is based upon the type view of the value on the right hand side of the assignment rather than that of the location on the left hand side. During the execution of the procedure with a parameter of type *director* the test will reveal that the creation type of the *e.vehicle* location is *Jaguar*, which is not a supertype of *car*, and so a failure will occur on the assignment.

5.2.2 Formalism

In formal terms the dynamic check ensures that whenever an update occurs to a location *i* from a view *j* then

$T_{\textbf{r-value}} (loc\ (i)) \leq T_{\textbf{creation}} (loc\ (i))$

The static subtyping rules also ensure that only supertype views of a location may be constructed:

$\forall_i . \forall_j . T_{\textbf{creation}} (loc\ (i)) \leq T_{\textbf{view (j)}} (loc\ (i))$

The combination of these inequalities gives the required invariant in the presence of assignment:

$\forall_i.\forall_j.T_{\mathbf{r\text{-}value}}\ (\mathrm{loc}\ (i)) \leq T_{\mathbf{view\ (j)}}\ (\mathrm{loc}\ (i))$

as required.

5.2.3 Modelling ability

This system allows the full range of subtype modelling as described by the principle of substitutability. All but the last of the desired operations described in Section 2.2 over the employee/car database may be described. In particular, the important class of operations disallowed by the view equality check may now be performed. An example from this database would be a procedure which took a collection of employees and a collection of cars and updated the cars belonging to a selection of the employees. In a database there would be an integrity constraint to ensure that directors would only be assigned Jaguars and there would be sufficient semantic information in the records of both the employees and the cars to ensure that an incorrect update did not occur. This would also guarantee the preservation of type security even though the types are not distinguished within this context.

5.2.4 Miscellaneous

In general this mechanism requires a full subtype check to be carried out whenever a location within a value is assigned to. The cost of this within a database system has not yet been investigated in detail but it could be severe. However, it may be possible to elide checks where the database integrity constraints provide sufficient semantic information.

5.3 Dynamic view check

The last mechanism is the most general which preserves type accuracy. All of the systems considered so far have preserved not only the invariant

$\forall_i.\forall_j.T_{\mathbf{r\text{-}value}}\ (\mathrm{loc}\ (i)) \leq T_{\mathbf{view\ (j)}}\ (\mathrm{loc}\ (i))$

but also the slightly more restrictive invariant

$\forall_i.\forall_j.T_{\mathbf{r\text{-}value}}\ (\mathrm{loc}\ (i)) \leq T_{\mathbf{creation}}\ (\mathrm{loc}\ (i)) \leq T_{\mathbf{view\ (j)}}\ (\mathrm{loc}(i))$

It is possible for the most specific view of a location to become a supertype of its creation type and if this occurs it may be possible for an update to occur with a supertype of the creation type without compromising type safety.

As an example, consider the program in Figure 8. This is similar to the incorrect program presented earlier, except for the update which occurs within the *sackMD* procedure.

In this program there is no breach of type accuracy. This is due to the fact that, when the update to *formerMD.vehicle* occurs, there exists no context in which the location denoted by *formerMD.vehicle* is viewed as a subtype of *car*.

```
type address is [place : string]
type thing is [name : string]
type person is thing with [addr : address]

type car is [model : string ; colour : string]
type Jaguar is car with [extraFuelTanks : int]
type employee is person with [employeeNo : int ; vehicle : car]
type director is employee with [vehicle : Jaguar]

let MD : director := director [
                name := "Ronald",
                addr := address [place := "St Andrews"],
                employeeNo := 1,
                vehicle := Jaguar [model := "Etype",
                                        colour := "red", extraFuelTanks := 1]]

let friend : employee := MD  !MD viewed as an employee

let sackMD : employee → void := proc (formerMD : employee) : void
begin
        MD := director [
                name := "Richard",
                addr := address [place := "Tentsmuir"],
                employeeNo := 2,
                vehicle := Jaguar [model := "Dtype",
                                        colour := "blue", extraFuelTanks := 3]]

        formerMD.vehicle := car [model := "model T Ford", colour := "red"]
end

sackMD (MD)
let three : int := MD.vehicle.extraFuelTanks            ! no error
let ford : string := friend.vehicle.model               ! still allowed
```

Figure 8 Use of a dynamically calculated view set

5.3.1 Failure capture

At the point of execution of the erroneous assignment in Figure 1, the type accuracy invariant becomes violated and the program would be terminated.

5.3.2 Formalism

The formal correctness of this system is trivial: the type accuracy invariant,

$$\forall_i . \forall_j . T_{\textbf{r-value}} (\text{loc } (i)) \leq T_{\textbf{view (j)}} (\text{loc } (i))$$

is used as a post-condition for any store update operations, and therefore any program which would cause the invariant to be violated will not be allowed to continue in its execution.

5.3.3 Modelling ability

A system such as this is able to model those cases within databases where entities possess different sets of attributes over time. For example, a person may become a student at some point in their life, at which point they possess an attribute of matriculation number. After they graduate they lose this attribute. This may be modelled by subtyping, by viewing the entity from a context where the extra attributes are no longer visible and by removing all access from any contexts where the more specialised view is available.

As an example in the employee/vehicle database model, imagine that a director resigns from the board of the company but continues to be employed. This may be modelled simply by removing the value corresponding to the director from the collection of directors, but by leaving the same value within the collection modelling all employees. After this has been done, no access to the extra fields modelled for a director is possible.

This kind of attribute loss may be modelled in any of the systems described which rely upon dynamic checking, but may not be modelled by any of the fully statically checked ones as the general collection of employees may not be modelled. There is a further problem however with the other dynamic models. It should now be possible for the ex-director to be assigned a vehicle which is not a Jaguar, and as the creation type of the ex-director's *vehicle* field is *Jaguar*, then an update with any other type of car will not be allowed. With this model, as *employee* is the most specific type with which the ex-director may be viewed, the update will succeed.

5.3.4 Miscellaneous

In general, the problems of implementing such a system are significant. Once again, however, it is possible that dynamic type testing may be avoided by the static assertion of integrity constraints in which case such a system may become more feasible.

5.4 No static constraint on assignment

So far only systems which include a static constraint on assignment have been considered. This constraint is that is the view type of the right hand side must be a subtype of the view type of the left hand side. As this static constraint applies only to the view type there may be cases where type accuracy is preserved when this constraint is not.

```
let jack : person := person [name := "Jack", addr := address [place :=
"Kilconquhar"]]

let MD : director := director [
            name := "Ronald",
            addr := address [place := "St Andrews"],
            employeeNo := 1,
            vehicle := Jaguar [model := "Etype",
                                    colour := "red", extraFuelTanks := 1]]

jack := director [
            name := "Jack",
            addr := address [place := "Kilconquhar"],
            employeeNo := 2,
            vehicle := Jaguar [model := "Etype",
                                    colour := "red", extraFuelTanks := 1]]

MD := jack
```

Figure 9 No static constraint on assignment

As a simple example, consider the program in Figure 9. The example performs a super to subtype assignment but still the system preserves type accuracy. In the final assignment statement, the view type of the right hand side is *person*, and that of the left hand side is *director*. In this case, however, the assignment is sensible and there is no loss of type accuracy: the assignment would succeed at the dynamic checks described in both Section 5.2 and 5.3. Notice that the dynamic check described in Section 5.1 relies upon the static subtype restriction and is therefore not applicable in this context.

In systems with explicit inheritance, it only makes sense to allow the assignment statically where the types on either side are related within the inheritance lattice. The language Modula-3 [CDG88] uses this rule for its classes, where subtyping is explicit.

5.4.1 Modelling ability

The failure capture and formalism for systems with no static constraints are the same as for the corresponding systems with static constraints. However, these systems have the extra ability to model update of a location with a value which is known to be appropriate, even if the view types of the location and the value are not in the subtype relation. An example of this has already been shown in Figure 9.

One possibility of particular interest with this system is that it may allow a value to be viewed as a more specific type than that with which it was created, but without losing its identity. In the employee/director database, it would be possible for a value to be created as an employee, then to have a Jaguar assigned to the vehicle field, and then subsequently to be placed in the collection of directors. The identity of the original value would not be changed during this sequence of operations.

5.4.2 Miscellaneous

In general, it is possible for any substitution operation to be performed without the loss of type accuracy whenever the view types of the two sides share a common lower bound in the type lattice. If the lack of restriction is applied orthogonally to a programming language, then some rather strange procedures, such as that shown in Figure 10, are valid. In Figure 10 the **with** construct is used to conjoin the type definitions of *bird* and *fish*. This procedure will work correctly when applied to any animal which has both gills and wings. Although this example requires dynamic checking, notice that as type accuracy is still preserved, it is still possible to write procedures which are statically guaranteed to succeed; only substitutions of supertypes or unrelated types imply any dynamic checking.

```
type fish is [gills : real]
type bird is [wings : int]

let silly : bird → fish := proc (birdAsFish : bird) : fish ; birdAsFish

type flyingFish is fish with bird
let oswald : flyingFish := flyingFish [gills := 2.3 ; wings := 3]

let os : fish := silly (oswald)
```

Figure 10 No static constraint on substitution

6 Conclusions

To maintain type accuracy, a system with both subtyping and assignment must prevent any dangerous update from occurring. Restrictions sufficient to preserve the type safety of such a system may be made in any of the following categories:

- substitution context limitation:

  limit the contexts in which substitution using inclusion may occur,

- substitution mutability limitation:

  model mutability within the type system, and restrict type compatibility in some appropriate manner, or

- substitution dynamic failure:

  accept that there may be a dynamic failure at time of update.

A number of possible mechanisms have been shown in each of these categories, none of which appear to be arbitrary, and all of which may be satisfactorily explained to a programmer. This classification should aid the designers of database programming languages in choosing a suitable mechanism to allow subtyping and mutability to coexist without compromising the overall safety of the system. What is clear is that the combination of subtyping and mutability should be approached with extreme caution.

Figure 11 sums up the modelling ability of each mechanism considered, according to the schema fragment shown in Figure 5. A tick in a column means that the mechanism is able to model features as follows:

1. a collection of all Jaguars
2. a collection of all directors
3. a collection of all cars (including Jaguars)

4 a collection of all employees (including directors) who will never change their cars
5. a collection of all employees (including directors)
6. a procedure which gives any employee's number
7. a procedure which swaps the vehicles of either two directors or two non-directors
8. a procedure which allocates a new car to a non-director (using the collection of all employees)
9. a procedure which allocates a suitable new car to any employee (including directors)
10. a procedure which demotes a director, and allocates a new vehicle which is not a Jaguar
11. a procedure which allocates a Jaguar to an employee who is then made a director

	1	2	3	4	5	6	7	8	9	10	11
Bounded universal quantification	✓	✓				✓	✓				
No subtyping over locations	✓	✓	✓								
Non-mutable views allowed	✓	✓	✓	✓		✓					
Dynamic view equality check	✓	✓	✓	✓	✓	✓		✓			
Dynamic general subtype check	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Dynamic view set check	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
No static constraint	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Figure 11 Summary of mechanisms

7 Acknowledgements

The work was supported by SERC Grant GR/F 28571 and ESPRIT II Basic Research Action 3070 - FIDE. We would also like to thank the following for careful proofreading and many constructive comments: Quintin Cutts and Graham Kirby at the University of St Andrews, and Tony Hosking, Tim Sheard and David Stemple at the University of Massachusetts at Amherst.

References

[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), 360-365.

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

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

[ADG89] Albano, A., Dearle, A., Ghelli, G., Marlin, C., Morrison, R., Orsini, R & Stemple, D. "A Framework for Comparing Type Systems for Database Programming Languages". 2nd International Workshop on Database Programming Languages, Oregon (1989). in **Database Programming Languages.** (Eds. R.Hull, R.Morrison & D.Stemple). Morgan Kaufmann Publishers Inc., Palo Alto, Ca, USA, 170-178.

[Alb83] Albano, A. "Type Hierarchies and Semantic Data Models" ACM SIGPLAN 83: Symposium on Programming Language Issues in Software Systems (San Francisco, 1983) 178-186.

[AM85a] Atkinson, M.P. & Morrison, R. "Types, bindings and parameters in a persistent environment". Proc of the Appin Workshop on Data Types and Persistence, Universities of Glasgow and St Andrews, PPRR-16, (August 1985),1-25. In **Data Types and Persistence** (Eds Atkinson, Buneman & Morrison) Springer-Verlag. (1988), 3-20.

[AM85b] Atkinson, M.P. & Morrison, R. "Procedures as persistent data objects". ACM TOPLAS 7, 4 (October 1985), 539-559.

[AM86] Atkinson, M.P. & Morrison, R. "Integrated Persistent Programming Systems". 19th International Conference on System Sciences, Hawaii, U.S.A., (January 1986), 842-854.

[BO91] Buneman, O.P. & Ohori, A. "A Type System that Reconciles Classes and Extents". Proc 3rd International Workshop on Database Programming Languages, Nafplion, Greece (August 1991) 175-186.

[Car84] Cardelli, L. "A semantics of multiple inheritance". In **Lecture Notes in Computer Science**. 173, 51-67. Springer-Verlag (1984).

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

[Car89] Cardelli, L. Typeful Programming. DEC SRC Report 45, May 1989.

[CBC90] Connor, R.C.H., Brown, A.L., Cutts, Q.I., Dearle, A., Morrison, R. & Rosenberg, J. "Type Equivalence Checking in Persistent Object Stores". 4th International Workshop on Persistent Object Systems, Martha's Vineyard, USA. (1990), 151-164.

[CDG88] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B. & Nelson, G. "The Modula-3 Report". DEC SRC Report 31, (August 1988).

[CM91] Connor, R.C.H. & Morrison, R. "Subtyping without Tears". Submitted for publication.

[CW85] Cardelli, L. & Wegner, P. "On understanding types, data abstraction and polymorphism". ACM.Computing Surveys 17, 4 (December 1985), 471-523.

[DCT91] Thomson, D.C. "Dennis the Menace and Gnasher" The Beano, 34, 7 (June 1991) 1-2.

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

[KC86] Khoshaftan, S. & Copeland, G.C. "Object Identity". Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon (September 1986), 406-416.

[MBC88] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. **The Napier88 Reference Manual**. PPRR-77-89, Universities of St Andrews and Glasgow (1989).

[Mil83] R. Milner "A Proposal for Standard ML". University of Edinburgh CSR-157-83 (1983).

[Str67] Strachey, C. "Fundamental concepts in programming languages". Oxford University, Oxford (1967).

[WZ88] Wegner P. & Zdonik S.B. "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like". In Proceedings ECOOP '88 – European conference on Object-Oriented programming, **Lecture Notes in Computer Science** 322, Oslo, Norway, (August 1988) 55-77.

[ZM90] Zdonik, S.B. & Maier, D. **Readings in Object-Oriented Database Systems** Morgan-Kaufmann (1990).