Type Equivalence Checking in Persistent Object Systems

Connor, R.C.H., Brown, A.L., Cutts, Q.I., Dearle, A.,[†]
Morrison, R. & Rosenberg, J.[¥]

University of St Andrews, Scotland
[†] University of Adelaide, Australia
[¥] University of Newcastle, Australia

email: {richard,ab,quintin,ron}@cs.st-and.ac.uk
al@adelaide.edu.au
johnr@nucs.nu.oz.au

Abstract

Two common methods of determining type equivalence in programming languages and database systems are by name and by structure. In this paper we will show that both mechanisms are myopic views of the type equivalence required for persistent systems. Methods of representing schema types within a persistent store will be discussed, and two possible implementations will be given. Finally discussion and measurements of the efficiency trade-offs for both representations will be presented.

1 Introduction

Type systems provide two important facilities within both databases and programming languages, namely data modelling and data protection. Recent developments in type systems have greatly increased their expressive power while retaining their traditional safety. These developments include parametric and inclusion polymorphism [CW85], abstract and existential types [MP85,CDM90], bulk data types such as sets, lists and arrays etc.[AM85], classes as type extensions [AGO89] and static constraint checking [SS89,SWB89]. One future goal for persistent programming languages is to develop a type system that will accommodate the structures required for both modelling and protection in less traditional database applications such as scientific programming, engineering applications and office automation, whilst also capturing the type description of more conventional database systems.

Our extended view of type systems allows the traditional database schema to be regarded as a type. We are concerned in this paper with how this schema type is manipulated efficiently by the persistent system. Central to this is how the schema type is represented within the persistent system and how type checking is performed using the schema type.

Traditional type checking within programs and type checking within the persistent store are rather different animals. Type checking within a program may entail a type checker in building a representation of the type, inferring some types, checking for the equivalence of the types and some compatibility checking on the types for coercions and subtype polymorphism, for example. Within the persistent store type checking is generally only concerned with checking the equivalence of two types for which representations already exist. For the moment we restrict this to an exact equivalence check and remark that subtype checking is somewhat more complex [ACP89].

Two common methods of determining type equivalence are by name and by structure. We will define these terms for clarity later in the paper and go on to show that both mechanisms are myopic views of the type equivalence that is required for persistent systems. To do this we will discuss the issues involved in schema type evolution, the distribution of the schema type in a distributed system, the merging of independently developed schema types, late binding to the schema type and how programs partially specify the section of the schema type of interest. We will then discuss the representation of the schema type within a persistent store, and present two suitable methods of representation. A comparison of these methods is given, along with measurements of the efficiency of both representations.

2 Models of Type Equivalence Checking

In database systems and programming languages, two methods of checking type equivalence are common. These are by name and by structure. The definitions of these two mechanisms given below are taken from [ADG89]. They are:

- in name equivalence, two values have equivalent types if the types share the same declaration, and

- in structural equivalence, two values have equivalent types if the types have isomorphic structures.

We will use these definitions to highlight the issues in this paper but note that most systems do not adopt such extreme positions for all their type equivalence checking. Many compromises can be made some of which will be exposed later.

It should also be noticed that the issue of static and dynamic checking is not of importance here. The separation between compile and run time can be obscure in a persistent system and does not affect the work involved in type equivalence checking. We are however still concerned with efficiency. We would prefer our type checking to be fast since depending upon the usage of the system it may have to be performed frequently. It is generally accepted that name equivalence is fast and structural equivalence is slow.

For these equivalence checking categories we will investigate type checking both within programs and within a persistent object store, which may be centralised or distributed.

2.1 Name Equivalence Checking

2.1.1 Name Equivalence Checking within Programs

The programming language Ada [Ich83] is a good example of a system that primarily uses name equivalence checking. An example is the definitions of the types *ANIMAL* and *VEHICLE* given below:

```
type ANIMAL is record
        Age : INTEGER;
        Weight : REAL;
end record;

type VEHICLE is record
        Age : INTEGER;
        Weight : REAL;
end record;
```

In structural terms, *ANIMAL* and *VEHICLE* define the same set of values from the value space, namely the labelled cross product *Age* : INTEGER x *Weight* : REAL. However, in name equivalence terms values of type *ANIMAL* are not type compatible with values of type *VEHICLE*. There are advantages and disadvantages to this.

One disadvantage is that anonymous types are not permissible in name equivalence and indeed Ada has to use an *ad hoc* mechanism to achieve the same effect. For example, it is not possible with strict name equivalence to write a procedure that will accept a parameter of a particular shape, such as a one dimensional array, even if it has a fixed size. The declaration,

```
procedure Add-elements (A : array (1..6) of INTEGER);
```

is achieved in Ada by having an anonymous type mechanism for the structure matching on the array, not by name equivalence. Such a disadvantage is only a minor drawback since it only takes a type definition to resolve the problem. For example

```
type INT_ARRAY_ONE_SIX is array (1..6) of INTEGER;
procedure Add-elements (A : INT_ARRAY_ONE_SIX);
```

At worst strict name equivalence is verbose. However, variable size arrays do cause problems for strict name equivalence checking since a different type name and procedure declaration are required for each size. Ada uses another *ad hoc* mechanism, that of unconstrained arrays, to overcome this.

Subtype checking, as used for inclusion polymorphism in object oriented languages, can be defined in name equivalence systems by explicitly stating the relation of a subtype to its supertype. For example

```
type LEGGED_ANIMAL isa ANIMAL with (No_of_legs : INTEGER);
```

This extends the supertype definition and ensures that *LEGGED_ANIMAL* is a subtype of *ANIMAL*. The mechanism is restrictive in that the structure of the inheritance hierarchy is explicit and costly to reconstruct.

With structural equivalence [Car84] one type is only a subtype of another if it has all the attributes of the supertype and possibly some more. The corresponding common attributes must be in the subtype relation themselves. To check that one type is a subtype of another, the structure of the two types constrained by the subtype relation must be checked. For example, consider the following declaration of *LEGGED_ANIMAL* :

```
type LEGGED_ANIMAL is record
        Age : INTEGER;
        Weight : REAL;
        No_of_legs : INTEGER;
end record;
```

In this case *LEGGED_ANIMAL* is a subtype of *ANIMAL* implicitly. This allows arbitrary construction of the inheritance hierarchy but requires a structural type check to validate that *ANIMAL* and *LEGGED_ANIMAL* are in the subtype relation.

Problems similar to the above occur in languages with parametric polymorphism where the type of the specialised form is deduced from the polymorphic form. This can be regarded as a type coercion or type inference but requires a structural check.

The major advantage of name equivalence checking is that it is efficient. The check involves ensuring that two types have the same declaration, which can be reduced to a single word comparison in most machines. This advantage goes a long way to explaining the popularity of the scheme within many programming languages.

2.1.2 Name Equivalence Checking in a Persistent Object Store

Name equivalence checking within a persistent store is based on a dictionary of type names which contains the schema. Programs producing persistent data extend the schema with the type of the data before placing the data in the persistent store. Programs using this data must reuse the type definition contained in the schema type to ensure compatibility of the types. The dictionary of type names may be one level, tree structured or even a graph depending on the mechanism used for names in the system. Tree and graph name spaces are essentially distributed in nature.

The manipulation of the schema may become a major efficiency bottleneck in these systems especially in the presence of concurrent access. Typical operations on the schema type are:

- adding a type definition,
- removing a type definition,
- altering a type definition,
- using type definitions, and
- merging type definitions.

Adding a type definition to the type schema only causes problems of name clashes within the dictionary. This is most acute in a flat name space but is not a serious problem since the name has to be added before the separate compilation of modules using the type can be performed. Changing the clashing name to an unused one and altering the code for the modules that share the type description solves the problem.

Removing a type definition from the schema type is more difficult since data of that type may exist in the persistent store. Removing the type definition would ensure that the data could not be used again with its original type. Only where the type is abstracted over, such as in polymorphism, could the data be used. If there is no type abstraction, then either the request to remove the type is denied, if data of the type still exists, or the data is removed with the type definition, although perhaps not immediately. This requires the ability to reach all programs and data that use a particular type, from the type schema itself. This may not be trivial where data is encapsulated within objects. One interesting aspect of removing types occurs with mutually recursive types and is similar to the problem of removal of values from mutually recursive classes in Object-Oriented Databases posed by Atkinson [Atk89]. If the type is removed what happens to its mutually recursive partners?

The type descriptions within the schema type are used by compilers to generate efficient code for the manipulation of the data of that type. For example, this may entail generating static offsets for indexing and means that altering the type requires recompilation of the programs and data that use that type. As with removing types, this requires the ability to reach all programs and data that use a type. More interesting, in altering a type definition the system is not concerned that the alteration is performed on the same type, since that is already established by identity, but that the alterations are compatible with the existing data. This requires a structural check.

Using a type description is equivalent to using a name. It depends upon two programs finding the shared name. Although this may be difficult in a large system, software tools such as browsers can be used to help.

As the use of the persistent object store grows, the schema evolves and a need arises for merging definitions. This may occur because mistakes have been made in defining two separate types that are logically the same and building half a system with each definition or it may occur in a distributed environment where separate universes are merged. There are a number of solutions to this problem.

The first point is that merging definitions in a name equivalence scheme requires user intervention. Since there may be many definitions that are structurally the same the system cannot decide automatically which of the definitions have to be aliased. Of course, when the user specifies that two types have to be aliased then the system must check that the types are compatible for all the existing programs and data. This involves a structural equivalence check.

There are two approaches to aliasing. One method is to recompile all the programs which use one type with the definitions of the other. Existing data poses a more severe problem, as it must be read as the old type and re-written as the new. The old type can then be removed.

The recompilation approach to definition merging is only feasible for small object stores since the time for recompilation in large stores may be prohibitive. A second solution to the problem is to alter the equivalence check itself. By using an indirecxtion in the type definition the old value may be overwritten with a new one without altering any of the references to the type.

The problem of name clashes also appears when separate schema types are merged. This is a problem of user perception of how the system works and not a technological problem. One of the names can easily be changed in the dictionary and all problems of referring to the correct type handled by the persistent address translation. The difficulty is that users do not know the new name and worse still the old one exists with a different definition.

Name equivalence checking lends itself to the partial specification of the type schema since only names of interest need be used within programs and not the whole schema.

Late binding of programs and data requires that when the program runs the data has an equivalent type to the one compiled in the program. This is performed by checking the persistent identifiers of the type names of the program and the data. Thus name equivalence checking in a persistent object store may be very efficient.

Examples of name equivalence checking over a single persistent object store can be found in the object oriented database systems ENCORE [SZ86] and 0_2 [BBB88] and the persistent programming language Galileo [ACO85].

2.2 Structural Equivalence Checking

Structural equivalence checking poses a different set of problems for the user and system architect from name equivalence. In particular, structural equivalence checking may be complex in terms of time and space and is not even known to be decidable in some cases [Rec90]. Just as we have shown in the previous section that name equivalence schemes require to use a structural check for some activities we will show that structural equivalence schemes can use a naming system to overcome some of the efficiency problems.

First of all we will describe structural equivalence in programs and in persistent stores, highlighting some of the advantages and pitfalls of the mechanism.

2.2.1 Structural Equivalence Checking within Programs

The language Napier88 [MBC88] will be used to describe our examples of structural equivalence since it uses only that form of type equivalence. The type definitions of *ANIMAL* and *VEHICLE* given in Ada earlier would be in Napier88:

> **type** animal **is structure** (Age : **int** ; Weight : **real**)
> **type** vehicle **is structure** (Age : **int** ; Weight : **real**)

These types are equivalent in Napier88 since the type declaration is only regarded as providing a syntactic shorthand for the set described in the type expression. The set of values in this case is the labelled cross product *Age* : **int** x *Weight* : **real**.

The freedom of this mechanism is that it allows type names to be used anywhere a type expression is valid and vice versa. This means that anonymous types, i.e. type expressions, may be used. For example, a procedure may be defined that returns the *Age* field of all records of type *Age* : **int** x *Weight* : **real**. This is given below.

```
let age = proc (x : structure (Age : int ; Weight : real) → int) ; x (Age)
```

In the above the identifier *age* is declared to be a procedure which takes a parameter *x* of type **structure** (Age : **int**;Weight : **real**) and returns an integer result. The body of the procedure is the expression *x (Age)* which is the Napier88 syntax for selecting the *Age* field of *x*. The result is the value of the *Age* field.

The procedure *age* will work for values of type *animal* and *vehicle* as well as any other aliases of the cross product type. Notice that this is not polymorphism nor is there any form of subtyping. It is merely structural equivalence.

Another example of the use of structural equivalence is given in the passing of procedures as parameters. The procedure *integral* is given below.

```
let integral = proc (f : proc (real → real) ; a,b : real ; no_of_steps : int → real)
begin
        let h = (b - a) / float (no_of_steps) ; let sum := 0.5 * (f (a) + f (b))
        for i = 1 to no_of_steps do
        begin
                a := a + h
                sum := sum + f (a)
        end
        h * sum
end
```

In this code fragment the identifier *integral* is declared to be a procedure that takes four parameters and returns a **real** result. It calculates the integral of a function between two limits, *a* and *b*, using the trapezoidal rule with *no_of_steps* intervals. *a*, *b* and *no_of_steps* are given as parameters. The first parameter is the function to be integrated. Its name is *f* and its type is specified by the type expression **proc** (**real** → **real**). *f* is a procedure that takes a **real** parameter and returns a **real** result. Any procedure of this structural form may be used.

Thus integral may be called by,

```
integral (sin, 0.0, 3.14159, 10)
```

or by

```
let quadratic = proc (x : real → real) ; (3.0 * x + 4.0) * (x - 3.0)
integral (quadratic, 1.0, 4.0, 30)
```

without introducing a common type name for *f*, *sin* or *quadratic*. They are all structurally equivalent by definition.

As mentioned earlier anonymous types also solve problems with array parameters, and structural equivalence facilitates implicit subtyping and implicit inclusion polymorphism, as well as specialisation in parametric polymorphism. These benefits must be balanced against the cost of performing the structural check. This can be substantial, although it is inexpensive where two types are quite different in structure. It is only where two complicated types are equivalent or nearly equivalent that the cost may be significant. Of course, as compiler writers have known for some time, if the user defines a type name and uses that alias instead of an anonymous expression in all cases, the equivalence check can be resolved by name. This possibility is usually checked for by compilers before a full structural check is attempted.

2.2.2 Structural Equivalence Checking in a Persistent Object Store

With structural equivalence checking in a persistent object store there is no requirement for a centralised type schema. Types are stored with the objects and the schema is effectively distributed with the objects themselves. Programs producing persistent objects place them in the persistent store along with a representation of their type. Duplicate type descriptions may occur but this is unimportant for equivalence checking purposes since it is done by structure.

Programs which bind dynamically to existing persistent data may employ two possible methods of operation. In the first, the program defines a type equivalent to that of the data in the persistent store that it wishes to use and the compiler assumes that this assertion is correct. A type check is made as the data is accessed at run-time to validate this assertion. This allows for very late binding of program and data. The second method uses a software tool to browse the persistent store and pick up the type from the data in the persistent store. This will then be included in the program automatically and the system proceeds as above. The advantage of the second method is that the type need not be written down, which may be a considerable saving where complex types are used. Another advantage of the second method is the speed of the equivalence check, since equality can most often be established by identity. This is not name equivalence, since non-equivalence must still be checked by structure, but it achieves the same efficiency in normal use.

Partial specification of the type schema is less easy in structural equivalence systems than name equivalence schemes since the whole structure of the type must be written down. Again this can be overcome by the second method of operation but there are some other solutions. Type **dynamic** of Amber [Car85] and types **env** and **any** in Napier88 are infinite union types with dynamic injection and projection operations. The type structure of any part of the schema need only be specified up to the limit of these infinite unions, which is convenient for partial specification of the type schema.

We will now consider the same operations of the type schema that we considered earlier for name equivalence.

Adding a type definition to the type schema causes no problems. Duplicate representations of types may occur but, as described later, this will only affect performance.

Removing a type definition from the schema is not possible explicitly. Since the schema is effectively distributed then the type may only be removed by garbage collection after all objects bound to that description of the type have disappeared.

Altering a type definition can be accommodated but only through the compiler changing both data and program simultaneously. However, the change is local and does not affect every value of a particular type, only the instances bound to that description of the type. Another method such as an IPSE is required to apply the change to the entire system. Most persistent systems already rely upon a mechanism such as reflection for this, which can also provide genericity [SFS90], browsing facilities [DB88,DCK89], data modelling facilities [CAA87], schema editing and query facilities.

Merging type schema is not a problem with structural equivalence since the schema is already distributed. Adding another part only adds to the distribution. No recompilation is required for this.

The major drawback of structural equivalence is that it is sometimes slow. Checks on vastly different types can be performed quickly. It is only where the types are complex and equal, or nearly equal, that the check may be costly.

There are a number of optimisations that may be made. First of all the compiler can generate code using the persistent identifier of a type identified by some browsing tool. The type environments of Napier88 and Type::Type of Quest [Car88] give a basis for this. In this case the structural check may be shortened to identifier equality without losing the desirable semantics.

Secondly when the pointers to different representations are found to represent equivalent types one of the pointers can be overwritten with the other. Subsequent checks on this pair of pointers will be fast. This method can however be unstable depending on which pointer is chosen for overwriting.

A third speed up is for an autonomous process to scan the persistent store combining pointers for equivalent types. A table of preferred identities can be constructed and thrown away at any time.

These efficiency measures speed up the check for equivalence to identity checks as in name equivalence. However they do not speed up checks for non-equivalence or for equivalence where identity is not assured. Indeed they slightly slow them down since the identity check is performed first.

2.3 A Universal Equivalence Checking Mechanism

Both structural and name equivalence type checking are myopic views of what is required for checking in persistent object systems. Name equivalence schemes require the structure of the data to be retained for code generation and checking in schema merging. Structural equivalence schemes may often achieve the speed found in name equivalence. This is perhaps the holy grail: to find a method of structural checking that is as fast as name checking while retaining its flexibility.

3 Implementation of Structural Type Checking

3.1 Type Equivalence Checking

In a structural equivalence type system, the types consist of sets defined over the value space of a language. Membership of these is defined by some properties of the values themselves. Values are usually of the same type only if they have the same set of operations defined over them. Type equivalence is therefore an implicit property of a value, and values do not need to be constructed with reference to a type definition. To decide type equivalence, a language definition must include a set of type rules. These define the universe of discourse of the langauge and allow the type of any value to be deduced.

The universe of discourse of a type system may be represented by the set of base types and the set of type constructors. Type constructors allow the derivation of new types from other types and perhaps some other information. Where the language is data type complete, the universe of discourse is infinite, consisting of the closure of the recursive application of the type constructors over the base types.

The structural type equivalence relation may be with a similar set of rules. For two types to be equivalent, they must be created with the same type constructor and in an equivalent manner, using types which are themselves equivalent. An equivalence rule must be defined for each different type constructor.

To perform structural type equivalence checking, it is necessary to build representations of types which contain sufficient information to establish the defined equivalence for each constructed type. An equivalence function which traverses two instances of such representations must also be defined. The essential feature of any representation type is that there exists a well-defined mapping from the value space of the representations to the type space of the language. It may be desirable in some systems for different values to represent the same type, as long as the equivalence algorithm used implements an equivalence relation which respects the semantics of structural type equivalence.

3.2 Representing Types

Any type is either a base type or a constructed type. Constructed types are a composition of other types, along with some information specific to the particular construction. This information could consist of, for example, field names in a record type or the ordering of parameters for a procedure type. In general, therefore, a type representation consists of three parts:

- a label, to determine which base type or constructor it represents
- the information specific to the construction of this type, if any
- a set of references to other type representations

The equivalence algorithm for a representation must check that, for any two representations, that the labels are the same, the specific information is compatible, and that the other types referred to are recursively equivalent.

For some type systems there is a requirement that the chain of references may be circular. This is the case in a type system with recursion, where circular references are used to achieve a finite representation. For example, the type of an integer list may be

rec type IntList **is structure**(head : **int** ; tail : IntList)

Also, to represent a type system which includes values of either universally or existentially quantified types, it is essential for any quantifier type to contain a reference to the type to which it is bound, to allow either inference or explicit specialisation to deduce the correct type equivalence rules of values with these types.

These circular references, although not increasing the conceptual complexity of type representations, are the source of serious problems with the efficient implementation of a structural equivalence algorithm.

3.3 Efficient Structural Checking

Types may be represented in a relatively straightforward manner, and a suitable equivalence algorithm is not hard to specify. There is a full discussion of this in [Con88]. However, there are two factors which can cause serious problems with the efficient implementation of structural checking.

A trend in modern programming languages, and particularly database programming languages, is to provide more and more sophisticated type systems which allow more program errors to be detected statically. This is currently pushing knowledge of static type checking to its limits, and there are even systems which need to employ theorem provers within the type checking system. Programmers are encouraged to provide the most detailed type specification possible, as this increases the chance of a programming error being detected before execution. As a consequence of this, type specifications may become extremely large and complex. It may be imagined that the size of a database schema specified statically as a type is considerable. In a system which performs structural equivalence checking dynamically, it must be possible to check types of this complexity without incurring an unacceptable overhead.

The problem of large representations is compounded by the fact that they may contain cycles. In general, an algorithm which traverses a potentially cyclic structure must check at each stage that its area of current interest has not been previously traversed. If this check is not made, then the algorithm cannot be guaranteed to terminate.

The check for cycles must be made on an attribute which is unique to each component of the representation, rather than to the type constructor it represents. This may be, for example, the identity of a node in a graph representation or the starting position within a string representation. The difficulty here is that there is only a small amount of information specific to a particular constructor, most of the important information of the type representation being resident in its topology. It may not be possible to define a useful ordering over the node instances for the purpose of a fast lookup. This depends on the chosen representation and the implementation language. If there is no good ordering, the major cost of the equivalence algorithm becomes a check for equivalent cycles, and its complexity is O (n^2) where n is the number of nodes. This is because during the traversal of the graph, itself of O (n), the cost of checking whether a node has been previously visited is itself O (n).

The performance of algorithms to check equivalence is crucial in a persistent system, as checking may frequently be required during the execution of a program. Complexity of an algorithm is perhaps more important than performance within a conventional system, as it is likely that a persistent system would have custom-built support for an appropriate algorithm. After some more general discussion of efficiency considerations, two different methods which achieve this are described.

3.4 Normalisation

It may be seen that there is a major tradeoff between the cost of constructing type representations and the cost of executing the equivalence algorithm. For example, strings which consist of definitions within a language's type algebra contain sufficient information to perform equivalence checking, but the checking algorithm is complex. As the construction of representations is a task performed during the static checking of the program, and equivalence checking is performed during execution, it is clearly desirable to put as much of the burden as possible into the building of the representations.

For example, consider a type system which includes a structure type which is a labelled cross product. Two such type constructions are considered equivalent if they are constructed over equivalent types using the same labels, but the order of the labels is not significant. Therefore,

structure(a : **int** ; b : **bool**)

and

structure(b : **bool** ; a : **int**)

are equivalent. In general, as the ordering of the fields is unimportant, representations may be constructed with the fields in any order. In this case, the equivalence algorithm must allow for this during its execution. The fields may however be rearranged by placing them in alphabetical order according to the labels. If this is the case, the equivalence algorithm may then assume that the ordering of the fields is significant. Thus the task of equivalence checking may be simplified at the cost of complicating the task of representation building.

In general, it is possible for many differently "shaped" representations to be constructed for equivalent types. For example, consider the equivalent types:

```
        structure( a,b : structure( c : int ) )
```

and

```
        structure( a : structure( c : int ) ; b : structure( c : int ) )
```

If the algorithm which constructs type representations is written naïvely, then the first of these definitions may result in what is, in some sense, a minimal representation of this type, whereas the second may contain duplicate components. For representations which contain cycles an infinite number of possible representations exist for any one type, although again there is only a single minimal representation.

A normal form is one in which no two component representations are equivalent to each other. The construction of a normalised representation may be highly expensive computationally, as it involves checking every component representation against every other one. This involves the execution of n^2 embedded equivalence checks, where n is the number of nodes. Balanced against this, for some classes of representation the equivalence algorithm for normalised representations may be substantially faster. This will be discussed in more detail later.

3.5 Representing types by graphs

In an implementation language which has a constructor type such as a record or structure, a graph representation of types is straightforward and elegant. It is highly suitable because of the recursive nature of type definitions and the requirement to have circular references between constructor nodes. This makes such representations simple to build and to decompose, and as such they are ideally suited for static type checking purposes. In one implementation of the Napier88 system the following representation type is used:

```
        rec type TYPE is
                structure( label : int ; specificInfo : string ; references : list[ TYPE ] )
```

This is sufficient to uniquely represent any type describable by the Napier88 type system using some straightforward mapping rules. The *label* field distinguishes the base type or constructor each node represents. The *specificInfo* field contains information such as fieldnames, concatenated together with markers to form a single string. The *references* field represents all references to other types from this type constructor. This has an implicit ordering which may be used as part of the type information where required.

Type equivalence is a recursively defined algorithm over this structure, and must check only for equality of the *label* and *specificInfo* fields, before recursively checking any representation in the *references* field. The following algorithm would work for type systems where cycles are not required:

```
        rec let eqType = proc( a,b : TYPE -> bool )
                a = b or                !** this means pointer equality (identity)
                (
                        a( label ) = b( label ) and
                        a( specificInfo ) = b( specificInfo ) and
                        eqList( a( references ) , b( references ) )
                )

        & eqList = proc( a,b : list[ TYPE ] -> bool )
                ( a is tip and b is tip ) or
                (
                        a isnt tip and b isnt tip and
                        eqType( head( a ),head( b ) ) and eqList( tail( a ),tail( b ) )
                )
```

As described previously, it may often be the case that the types being checked have the same identity. In this case the equivalence is detected immediately, otherwise the full structural check is necessary. Notice that the test for identity is also performed recursively, which optimises the case of two different representations sharing components.

When the possibility of cyclic structures is introduced, it is necessary to take further steps to ensure the termination of the algorithm for equivalent types. This is done by keeping a note of all pairs of nodes that are traversed in a "loop table". Before any pair of nodes is traversed, a check is made to see whether the same pair has already been encountered. If they have, then either the full recursive check over these nodes has already been performed, or else is in the process of being performed. If the check has already been performed, then the nodes must be equivalent, otherwise the algorithm would have already been terminated with failure. In the case where the test is still in the process of being performed, these nodes may safely be assumed to be equivalent. If they turn out to be equivalent then the assumption is correct and re-traversal of the loop has been avoided. If they turn out to be non-equivalent then the algorithm will in any case end with failure from another branch of the recursion.

The new algorithm looks like this:

```
rec let eqType = proc( a,b : TYPE -> bool )
a = b or
in_loop_table( a,b ) or
begin
        add_to_loop_table( a,b )
        a( label ) = b( label ) and
        . . .
```

The use of the loop table not only ensures termination in the case of a cycle in the graph, but also prevents multiple traversals of a shared component within the graph. This adds to the efficiency of the algorithm.

With this representation, recording and looking up pairs of nodes in the loop table may cause a performance problem, as no suitable key is readily available to use for indexing. This would result in long lists of pairs being searched for an identity match. This can be simply solved by introducing an extra field into the node structures. When each type representation is created, this field is initialised with a value which may be used as a key for the node. Each pair of nodes, as encountered, is now tabulated using one of these keys on the first traversal, and the cost of checking for cycles is no longer significant. Another possibility is to use a pseudo-random number instead of a unique key, and to use a hashing algorithm based on this.

The reason for using pseudo-random keys is that the hash table may be preserved between executions of the equivalence function, and will then act as a memo table for all pointer pairs which are compared more than once. The pseudo-random keys reduce the possibility of hash clusters forming.

This persistent hash table has the interesting feature that it is not required to preserve the correctness of the algorithm, and so may be re-initialised at any time. It is important also to note that should the equivalence algorithm fail, all nodes added during that execution must be removed. For this reason, a "shadow-copied" table is used, which may be either preserved or restored depending on the outcome of the equivalence test.

The use of these techniques allows the check for cycles to be performed in constant time, and so the checking algorithm may achieve complexity of O (n) where n is the number of nodes.

3.6 Representing Types by Strings

There are many possible ways of representing types by strings. One possibility is to use strings which consist of type definitions within the type algebra of the programming language, which would normally be sufficiently powerful to provide a representation for any describable type in the language's universe of discourse. However, for a sophisticated type system, any equivalence algorithm over such representations would be inefficient.

A normalised string representation is more useful. An ideal transformation from types to strings would be canonical, so that there is a reversible mapping from strings to types. In this case, the type equivalence relation may be modelled by string equality. This may be implemented in a computer by a block comparison, an extremely fast operation on most machines.

A constructive proof that a canonical string form exists for a particular type system, although not necessarily difficult, is beyond the aims of this paper. A method of construction will instead be outlined.

The method relies upon the assumption that the graph representation and equivalence algorithm outlined above are sufficient to model type equivalence in the system in question. Firstly an algorithm will be described which produces a normal form of any such graph. Normal graphs are canonical representations of types. Another algorithm will then be described, which maps graphs to strings. This mapping is demonstrably reversible, and therefore gives the desired result of canonical strings.

3.6.1 Normalising Graphs

The condition for a graph to be normal is that no two nodes of the graph represent equivalent types. Therefore the type equivalence algorithm may be applied to any two nodes within such a graph, and will always fail unless the nodes have the same identity. The algorithm we will describe to map a graph to its normal form operates by copying the graph, but mapping any equivalent nodes in the original graph to a single new node.

This algorithm relies upon a data structure similar to the loop table of the previous algorithm. In this context we will describe it as a "memo table". Each node traversed is placed in the table as before, but this time it is used to memoise the result which has already been, or is currently being, calculated for the normal form which represents the node. The algorithm to produce the normal form of a node checks in this table to see whether it has previously produced, or is in the process of producing, a normal representation for an equivalent node. The check is thus based upon the equivalence algorithm previously described, rather than simple identity.

The algorithm is as follows:

1. Construct a new memo table

2. To copy a node, first check in the memo table to see if a node with an equivalent type has already been copied. If it has, return the corresponding node stored with it. Otherwise,

3. Create a new node, without filling in the fields.

4. Add to the memo table a pair consisting of the node being copied and the new "dummy" node.

5. Fill in the fields appropriately, including recursive use of this algorithm to fill in the component types.

Notice the way that each new node must be created as a "dummy" so that each pair of identities may be added to the memo table before any recursive calls are made. Notice also that because the test applied to the memo table is equivalence, rather than identity, any nodes in the original graph which are equivalent will be mapped to the copy of the first of these nodes which is traversed during the copy algorithm. The resultant graph is therefore normal. For any type system which may be correctly represented using this graph representation, there exists only a single normal form which represents each type. Therefore there exists a reversible mapping between normal representations and types, and so this representation is canonical. This is achieved at the cost of executing the equivalence algorithm over every pair of nodes within the original graph.

3.6.2 Mapping to Strings

The following algorithm maps graphs to strings. It again uses a memo table, this time to provide unique names for any types which occur more than once in the representation. Again, this deals both with cycles and with shared components in the graph representation.

> *startSymbol*, *endSymbol*, and *separatorSymbol* are mutually distinct characters which do not occur within the strings found in graph nodes.

1. Construct a new memo table
2. Initialise *nextMarker*, a procedure which produces a deterministic series of unique strings, a different string being produced on each call. These strings consist of characters which do not occur within the strings found in graph nodes, and do not contain the characters *startSymbol*, *endSymbol*, and *separatorSymbol*
3. First check in the memo table to see if the node with this identity has already been traversed. If it has, return the corresponding string stored with it. Otherwise:
4. Store the node in the memo table, associated with the result of calling *nextMarker*.
5. The result string is the concatenation of:
   - *startSymbol*
   - The node's *label* field in string format
   - The node's *names* field
   - The concatenation of the recursive application from Step 3 of this algorithm to any component types
   - *endSymbol*

The reader should be convinced that the above algorithm both terminates and produces a unique form for any graph. The unique markers used within the string for loops in the graph depend upon the traversal order of the graph; as the component types within any node are ordered as part of the normalisation process this order is always significant type information. The n^{th} marker produced by the *nextMarker* symbol corresponds to the n^{th} *startSymbol* which occurs within the string, and so these markers act as explicit references within the string. The string produced may thus be regarded as a normalised set of mutually recursive type definitions.

As these string forms are canonical, the type equivalence relation is implemented by string equality. The important result is that a flattened form may be found, which is implemented by a block comparison. This operation is already optimised in much conventional hardware, and so such a representation may be highly suitable for a prototype persistent system.

3.7 Comparison of Graphs and Strings

3.7.1 Speed

As has been shown, linear complexity can be achieved for equivalence algorithms over both graphs and strings. This would imply that either representation is reasonable to build into a persistent system, as no dramatic slowdown would be associated with a programmer using more complex types in a program.

However, although the complexity may be the same, the hidden constant may be significantly different. The block comparison associated with strings would be faster than the graph traversal, even if both systems were constructed in hardware. In particular, conventional hardware is already optimised to perform block comparisons, and so the string representation should be substantially faster on an existing machine.

3.7.2 Space

Due to its recursive nature, the graph equivalence algorithm uses more space during execution than string equivalence. This temporary space however is not expected to be a significant cost in a persistent system, especially compared with the permanent space required for the type representations.

The overall space occupied by a single representation is significantly greater for graphs than strings. This is because each node in a graph carries the overhead associated with a persistent object, whereas a string may be implemented as a single object. Also, references to other nodes are persistent identifiers, which may be large depending upon the implementation of the persistent store. As the amount of information contained within any single node is relatively small, these overheads will be the major space cost of a graph representation.

The strings however require contiguous space, whilst the graphs do not. There are advantages on both sides of this. For large types, comparing strings may cause a large amount of volatile memory to be used up at one time, whereas the graph nodes may be fetched from non-volatile store in pairs if necessary to use only a small amount of space. On the other hand, fetching the graph requires a large number of object faults, whereas only one is required for a string. The importance of these considerations depends upon the implementation of the persistent store.

As already stated however dynamic checking may be worthy of customised implementation, and it would be expected that both string and graph type representations would be adjusted to have similar characteristics using clustering, compression and fragmentation techniques as appropriate.

3.7.3 Sharing

So that the block comparison may be made over strings, all of the references to other types are converted so that they may be interpreted only within the context of the string. This means, unlike the graph representation, that sharing of common component type representations is not possible. This has serious implications for both time and space complexity.

For example, a program may require two values of related types from the store. This is common in persistent programming, where one procedure may generate an object of a complex type and others may use it in different ways:

```
type aType is ....

let generator = proc( -> aType ) ; ...
let user1 = proc( aType -> int ) ; ...
let user2 = proc( aType -> bool ) ; ...
```

Using graph representations, the type representations of these three procedures may use the representation already constructed for *aType*, and so only a few extra graph nodes are required to represent the more complex types. Using strings, however, complete new strings must be constructed for each procedure type, which duplicate all of the information already in the string constructed for *aType*, as the context-sensitive references may be different. This is because strings are a truly anonymous representation, whereas graphs contain implicit naming information in their store addresses. The difference in space may be significant for programs which use a complex type in many different ways.

Another aspect of sharing components is that any memoisation performed over these components then carries over all types which use them. In the above example, it would be common practice for one program to define the three procedures and place them within the store and for another to subsequently access them. In this case, the memoisation performed by the graph equivalence algorithm will mean that the structural equivalence check is only performed once on the type *aType*, whereas the full structural check is required for each different procedure if a string representation were used. Again, this is a substantial saving for a large class of programs.

3.8 Measurements

To give an idea of the expected performance of these algorithms, we include some measurements taken from the Napier88 system. We have made measurements to indicate space overhead, performance, and complexity of the different schemes.

All measurements were made with the type of a Napier88 abstract syntax tree. This is an extended and revised version of PAIL [Dea87], and is a large, mutually recursive type, consisting of around one hundred and forty definitions.

The measurements of performance are hard to quantify, as they are highly dependent upon the implementation of the Napier88 system within which they were made. Suffice it to say that the best figures we have achieved for checking independently prepared versions of this type are at the rate of several per second for a graph representation, with a substantial speedup to several hundred per second for a string representation.

The complexity measurements confirm the deduction that complexity at least as good as linear may be achieved for checking graph representations, with a suitable size of hash table. It is a reasonable assumption that a large enough table may be employed, as this is a fixed overhead per system. As explained, the table may be re-initialised if it becomes too large. In the case where the table may be large enough to contain all types used within the system, then the resulting memoisation achieves the same efficiency as name equivalence checking.

3.8.1 Space

The graph representation of the Napier88 PAIL type consists of 413 objects, with an average of just under nine words per object. The total size of this graph is 14,466 bytes. The string form of the graph is 2,206 bytes long.

These figures are all taken from relatively naïve representations; we have made no serious effort to compress the representations.

The benefits of space saving by sharing are hard to quantify, as they depend very much upon the manner in which the types are used. However, in our use of the abstract syntax tree type in building a Napier88 compiler, we found that the type is used in 276 different contexts. Each context requires a different string for its representation, as previously explained, but a graph may be shared by different contexts. Even if the system is fully optimised and only a single representation is used for each type, the total amount of store used to represent this type by strings is therefore more than 600,000 bytes, as oppose to a constant 14,466 bytes for the graph representation. Furthermore, each of these 276 representations requires at least one full structural check, whereas the first check using the graph representation acts as a memo for any other check performed while the system is running.

4 Conclusions

Type systems in persistent programming languages are assuming an increasingly important role, and the traditional database schema is now commonly regarded as a type. This leads to a requirement for the efficient manipulation of types in a persistent system to allow provision of the facilities traditionally found in DBMS for schema editing, use and evolution. We have chosen one aspect of schema manipulation in this paper, that of type equivalence checking, and have described the use of the common methods of name equivalence and structural equivalence.

We have shown that while name equivalence schemes are easier to implement and are more efficient they still have to use structural checks to provide important facilities such as schema merging. On the other hand structural equivalence, generally more flexible and less efficient, can often achieve the same performance as name equivalence.

Given that the efficiency of name equivalence is adequate for our needs, we have concentrated on how to improve the performance of structural equivalence checking. We have described how such checking is performed, and shown that there is a balance between constructing efficient representations of types in terms of store and the speed of the equivalence algorithm in comparing two representations.

We have shown how types may be represented by strings and by graphs, highlighting the difficulties in their construction and use. Some preliminary measurements are presented and our main conclusion is that where the type schema is large and involves the sharing of types, the graph representation will be much more efficient in terms of space. It may however be slower in terms of speed of checking depending on its use within the persistent store.

5 Acknowledgements

We would like to acknowledge funding from the following: SERC grant GRF 02953; Esprit II Basic Research Action 3070 - FIDE; SERC Visiting Fellowship GRF 28571 which allowed John Rosenberg to visit St Andrews for a year, and the UK DTI Object-Oriented Awareness Initiative. We would also like to thank Mike Livesey of St Andrews University for his many constructive criticisms.

6 References

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

[ACP89] Abadi, M., Cardelli, L., Pierce, B.C. & Plotkin, G. "Dynamic Typing in a Statically Typed Language". DEC SRC Report 47, (June 1989).

[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". *Proc 2nd International Workshop on Database Programming Languages*, Oregon (June 1989), pp. 203-212.

[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.

[AM85] Atkinson, M.P. & Morrison, R. "Types, bindings and parameters in a persistent environment". *Proc. 1st 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.

[Atk89] Atkinson, M.P. e-mail barrage on O-O classes and deletion. (1989-90).

[BBB88] Bancilhon F., Barbedette G., Benzaken V., Delobel C., Gamerman S., Lecluse C., Pfeffer P., Richard P. & Valez F. "The Design and Implementation of O_2, an Object Oriented Database System". Proc. 2nd International Workshop on Object-Oriented Database Systems, West Germany. In **Lecture Notes in Computer Science 334**. Springer-Verlag (1988), pp. 1-22.

[CAA87] Cooper, R.L., Atkinson, M.P., Adberrahmane, D. & Dearle, A. "Constructing Database Systems in a Persistent Environment". 13th VLDB, Brighton, UK, (September 1987), pp 117-126.

[Car84] Cardelli L. "A Semantics of Multiple Inheritance",*In Semantics of Data Types,* **Lecture Notes in Computer Science 173**. Springer-Verlag (1984) pp 51-67.

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

[Car88] Cardelli, L. "Typeful Programming". *1st European Conference on Extending Database Technology*. In **Lecture Notes in Computer Science 303**. Springer-Verlag (1988).

[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.

[Con88] Connor, R.C.H. "The Napier Type Checking Module". Universities of St Andrews and Glasgow. PPRR-58-88 (1988).

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

[Dea87] Dearle, A. "A Persistent Architecture Intermediate Language". PPRR-37-87, University of St. Andrews. (1987).

[DB88] Dearle A. & Brown A.L. "Safe Browsing in a Strongly Typed Persistent Environment". The Computer Journal 31,6, (December 1988), pp. 540-545.

[DCK89] Dearle, A., Cutts, Q.I. & Kirby, G. "Browsing, Grazing and Nibbling Persistent Data Structures". *3rd International Conference on Persistent Object Systems*, Newcastle, Australia (1989), pp 96-112.

[Ich83] Ichbiah et al., The Programming Language Ada Reference Manual. ANSI/MIL-STD-1815A-1983. (1983).

[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).

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

[Rec90] **type** AnyArray[t] **is variant**(simple : t ; complex : AnyArray[**array**[t]])

[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).

[SZ86] Skarra A. & Zdonik S.B. "An Object Server for an Object-Oriented Database System", *Proc. International Workshop on Object-Oriented Database Systems*, Pacific Grove California (September 1986) pp 196-204.