A Framework for Comparing Type Systems for Database Programming Languages

Antonio Albano[1], Alan Dearle[2], Giorgio Ghelli[3], Chris Marlin[4],
Ron Morrison[2], Renzo Orsini[3] & David Stemple[5]

[1]Dipartimento di Matematica e Informatica
Università di Udine
Via Zanon 6 - 33100 Udine
Italy.

[2]Department of Computational Science
The University of St Andrews
St Andrews
Scotland.

[3]Dipartimento di Informatica
Università di Pisa
Corso Italia
40 - I-56100 Pisa
Italy.

[4]Department of Computer Science
The University of Adelaide
Adelaide
South Australia
Australia.

[5]Department of Computer Science
University of Massachusetts
Amherst Massachusetts
U.S.A.

Abstract

Several proposals have been published in recent years for database programming languages (DBPLs), many of which have been object-oriented. Our goal in this paper is not to argue for or against specific solutions, but simply to provide a framework for comparing certain critical points of type system design. This framework may be used in the description of a DBPL. It is our hope that the framework will promote clear communication among designers and developers of DBPLs.

1 Introduction

At the Second Object-Oriented Database Workshop [OO88], the first author posed a set of questions as a way of urging people to be clear when talking about types and/or classes. This brief paper expands on those questions in an effort to provide a framework for describing type systems that can be used to compare programming languages in a meaningful manner. We have divided the questions to be asked into five areas. The first is discussed in Section 2 and has to do with the nature of the type system.

Section 3 deals with the expressiveness of the type definition capabilities and lists some of the type constructors that could appear in a type system. This section also addresses the limitations on expressibility in the type definition language and the constraints that can be expressed in the language, both explicitly and implicitly. It is also this section which covers polymorphism in the type system.

Section 4 is devoted to details of the value space. The issues addressed in this section include questions such as whether each type has equality defined on its values, whether functions can be values in a type, and whether there are to be objects as well as values in types.

Section 5 deals with relationships between types, covering the question of type equivalence and how concepts such as abstract types and object types relate to each other. In Section 6, issues concerned with subtypes of types are explored, such as whether subtypes arise implicitly or explicitly, and the nature of subtyping rules in the type system.

The notion of a class as a type concept is addressed in Section 7, and the database aspects of the type system, such as persistence, are dealt with in Section 8. After some miscellaneous issues in Section 9, a few brief conclusions are presented in Section 10.

We have made an attempt to avoid being normative in presenting these questions. Similarly, we have mostly avoided definitions which may over constrain the utility of the framework. Our aim is to give a framework that facilitates good communication among the designers and developers of database programming languages, rather than to prescribe or proscribe certain sets of features of type systems. Our hope is that we will make it easier to compare different languages with respect to the semantics of their types; this is something which has been difficult to do in the past.

2 The Nature of the Type System

2.1 Does the language have a type system?

The type system of a language is the manner in which the set of values over which a computation may range is partitioned into subsets, the types, such that only values within a particular type may occur at a particular point in the computation specified in the language. A language without such a property (or, which is equivalent, with just one type) is called *type-free*. An example of a type-free language is the λ-calculus.

2.2 Is the language strongly typed? Is type checking static or dynamic?

A language is strongly typed if all computations are checked for type errors. A strongly typed language is statically checked if all type errors are discovered at compile time, and dynamically checked if type errors are only discovered at run time. There may be other times, such as link time, when checking may be performed. A mixture of checking times is also possible.

2.3 How is type checking used in data definition and for operations?

Type checking is intended to signify a check made on a program to determine if types are defined and/or used in a consistent manner. The *type checker* is the program which performs this task. There are two principal forms of checks, one which concerns only the definition of database schemata, and another which concerns the use of data:

- **Checks for database schema well-formedness:** In this case the data definitions are checked for consistency. This is usually the case of a data definition language (DDL) of a database language which deals with data definition.

- **Checks for correct utilisation of values:** In this case, all computations in the program are checked to detect whether a value appears in an improper place in a computation. In general, for each type T, there is a set of operations O_T that can be applied to its values; a value of type T is misused with respect to its type (i.e., a *type error* occurs), if an operation which does not belong to O_T is applied to it.

2.4 Is the type system used only for error checking or also for the specification of implementation details?

Type systems evolved in programming languages as a means of specifying implementation details, such as the layout of storage. In certain languages, such as typed functional languages, this role has diminished. In other languages, a major motivation is that of data modelling.

2.5 If the previous questions do not fully cover the nature of the type system, what is missing from the description of the system?

3 Expressiveness

The expressiveness of a type system concerns the following aspects:

- The set of primitive types offered (integers, strings, booleans, etc.) and type constructors (records, arrays, functions, unnamed types, abstract data types, etc.) with the associated operations, such as constructors, selectors and iterators.
- The kind of polymorphism of the system, e.g. the mechanism used to express the fact that a function can be legally applied to all values belonging to a family of related types.

These points are explored in the following sections.

3.1 What primitive types and type constructors are available in the language? Are there restrictions on the combinations of type constructors or the form of recursion?

Here, we use the terminology of Cardelli and Wegner [CW85] if they discussed the constructor. Among the type constructors which can be found in programming languages, we have:

- Primitive types: string, integer, boolean, floating point, bit string, trivial type (one element type), enumerated type.
- Cross product (pair, tuple).
- Record (labelled cross-product).
- Discriminated union.
- Variant (labelled disjoint sum).
- Array.
- List.
- Set.
- Sequence.
- Function.
- Reference.
- Mutable value.
- Predicate subtype (as in a subrange of integers – it takes a type and a predicate on elements of the type and produces a type having only those elements of the input type that satisfy the predicate).
- Parameterisation (abstraction over an expression defining a type to define new type constructors, such as tree of something or stack of something).
- Abstract data type (selective hiding over a type definition).
- Recursive type definition, such as:

rec type Department := [Name: string **and**
Employees: **seq** Employee]
and Employee := [Name: string **and** Dept: Department]

3.2 What kinds of polymorphism are supported by the type system?

Polymorphism means that a single function can act on values of a well defined set of related types. There are two popular kinds of polymorphism:

- **Universal polymorphism,** which has two forms:

  - *Inclusion polymorphism* means that a subtype relation is defined on types such that if B is a subtype of A, any function that can be applied to A can also be applied to B.

  - *Parametric polymorphism* means that a function can have an implicit or explicit type parameter which determines the type of value parameters and the type of the result. It is called implicit polymorphism when the type parameter is deduced from the type of value parameters, and explicit polymorphism when the type parameter must be specified explicitly.

- **Ad hoc polymorphism or overloading,** which occurs where a function acts differently for each allowable type. The arithmetic operator "/" (divide), which acts on integers and reals, is an example of an ad hoc polymorphic operator.

4 Types and values

This section deals with some properties of the relationship between values and types.

4.1 What are the properties that are possessed by values of all types? Is there a concept of first class values? What are the properties that define first class values? Do all types have these properties?

Obviously, not all values of all types can be treated the same. However, there are normally operations and properties that are shared by many if not all types. These often include the ability to be bound to identifiers, stored in memory locations, passed as parameters to functions and returned as results. These properties can be used to define the concept of *first class values*.

4.2 Are there values that can be typed that have special properties not shared by all values? What are they and how are they special? Among these might be functions, mutable values and types themselves.

In some languages functions are first class values of a specific family of types: the function types. In other languages, there is a way to define, store, identify and use ordinary (first class) values and another way to define and use functions.

The same situation might exist for mutable values. If the mutability of a value can be controlled in a type system, is this control explicit or implicit? If implicit, what structures include it implicitly? If explicit, with what constructors can it be included?

4.3 Does the type system allow the construction of objects?

We regard an object as an entity with a local state and equipped with a set of operations. An object has an "identity" which is independent of its state.

4.4 Is every value an object, or do both objects and non-object values exist? Are objects first class values? What is the exact distinction (if any) between objects and first class values in the language? How do types of objects fit into the type system?

4.5 What is the semantics of the equality relation?

Among the different possible semantics for equality we have:

- **Typed structural equality:** two values are equal if the same sequence of legally typed selectors applied to either of them will give the same result.
- **Equality based on sameness:** some values can have an "identity" which is independent of their value; these values are often called objects. Sameness tests if two expressions evaluate to the "same" object.
- **Low-level structural equality:** two values are equal if their internal representation by the run-time support is equal.
- **User defined equality:** some languages allow the user to define equality.
- **Others.**
- **Mixed equality:** different combination of the preceding ones are possible, and present in some programming languages.

4.6 Is there a concept of null value? Are there different kinds of null values? How do they interact with the equality relation?

5 Relationships among Types

In this section, we aim to elicit the properties of type interrelationships, such as those involved in type equivalence, subtyping and inheritance.

5.1 What is the nature of type equivalence?

A type equivalence rule is used by the type checker in determining whether a value has the correct type for its context. Type equivalence rules are important and complicated where program and data are separately prepared. There are two common forms of type equivalence:

- In **name equivalence**, two values have equivalent types if the types share the same declaration.
- In **structural equivalence**, two values have equivalent types if the types have isomorphic structures.

5.2 Does the language offer abstract, "concrete", and "unnamed" types?

Abstract types permit new types to be defined by specifying a representation type and the new type's operators, i.e. a set of operations which take as parameters and/or give as results values belonging to this new type. The representation type is hidden in an abstract type. A "concrete" type definition is an association between a name and a type expression which makes that name completely equivalent to that expression; so a concrete type is not a new type but is a new denotation for the representation type. A language offers "unnamed" types when a type expression can be used in situations where a type identifier could appear, for example to denote the type of the parameters of a function, the type of the fields of a record, and so on. As a consequence, an abstract type is different from any other type, as well as from its representation type, while a concrete or unnamed type is equal to its defining type expression.

Another possibility for defining new types is the semi-abstract type: the new type is different from any other type and from the representation type but it inherits the predefined operators on the representation type. Consider for instance, a semi-abstract type Person represented as a record type. In this case, the selection operators on the representation record type can be used to extract information from values of type Person.

5.3 How are abstract types related to object types?

Object types may be a special case of abstract data types. Alternatively, they may be the only kind of abstract data type that can be built in the language or defined through an unrelated mechanism.

5.4 Does the language offer a module mechanism and, if so, what are its features? Are modules related to abstract data types? Are modules realised through a type constructor or are they a concept unrelated to the type system?

Abstract types and modules are related as they both serve a software engineering function of detail hiding. They both provide "encapsulation", the insulation of an “interface”, i.e. the means of interacting with a module or abstract type, from a “body”, which is the implementation of the module or of the abstract type definition. In addition to data encapsulation, modules are used as unit of recompilation when the implementation of the operations in the body is modified.

6 Types and subtypes

6.1 Is the subtype relation defined implicitly or explicitly?

In general, there are two ways in which a language can define a subtype relation among two types A and B:

- **implicitly:** types A and B are defined independently, but the *subtyping rules* specify that one is a subtype of the other, or
- **explicitly:** type B is specified to be a subtype of A.

This issue is related to the type equivalence relation, as the first choice is similar to structural equivalence and the second to name equivalence. Both the possibilities can be present at once in a language.

6.2 What subtyping rules are adopted in the type system? Is there any notion of closure in subtyping? If so, are subtypes restricted to closed subtypes, or can closure be specified and enforced or verified?

By closure, we mean the following: if B is a closed subtype of A, then any operation that accepts values of type A and produces values of type A will produce a value of type B if applied to B type values.

6.3 Can redefinitions be made going down the subtype hierarchy? If so, what may be redefined and under what control?

For example, in object-oriented languages it is common to allow the redefinition of methods when declaring a subtype. This may include redefinition of the type and semantics of the method.

6.4 Is the subtype graph single or multiple? How are name conflicts resolved?

A new type can be defined as an immediate subtype of only one type (single hierarchy), or as a subtype of several types (multiple hierarchy). In both cases, any subtype can have multiple supertypes, in the first case because the subtype relation is transitive. Of course, when subtypes are inferred and not explicitly defined (see 6.1), a multiple hierarchy results.

A name conflict occurs when the same attribute is defined differently in more than one supertype. A common approach is to solve the conflict on the basis of the order of supertype specification.

7 Classes and subclasses

7.1 Does the language have a concept of class? If so, what is it?

One possible definition is for a class to be characterised by the type of its elements, and by the set of the elements of that type currently in existence.

7.2 How are a class and the type of its elements related?

When a database language has a type system in the sense previously defined, it is important to distinguish between the definition of the classes, i.e. sets of values which populate the database, and the definition of types. In fact types are sets of all possible values, while classes are sets of actual values, i.e., values currently present in the database. In many languages a class and the type of its elements are defined through a single construct and sometimes are given only one identifier, while in others the two concepts are clearly distinguished. Another related issue is whether all the values built in a program which belong to the type associated to a class automatically belong to the class or whether an explicit insertion operation is needed.

7.3 Does the language have a distinction between subtypes and subclasses?

This question makes sense in languages with classes and types as previously discussed. When there is a separation between these two concepts, there could be a concept of subclass in addition to that of subtype. A subclass concerns the extensional aspect of the *isa* relation: if we are interested both in Persons and Females, we have two different and essential facts: the type Female is a subtype of the type Person, because all the possible Females are a subset of all the possible Persons, and the class Females is a subclass of a class Persons because Females is a subset of all the actual Persons at all times.

In systems where the subclass relation is distinct from the subtype relation, the next four questions mimicking 6.1 to 6.4 need to be answered.

7.4 Is the subclass relation defined implicitly or explicitly?

7.5 What subclass rules are adopted in the type system?

7.6 Can redefinitions be made going down the subclass hierarchy? If so, what may be redefined and under what control?

7.7 Is the subtype graph single or multiple? How are name conflicts resolved?

7.8 Does the language have single or multiple superclasses? Is it the case that the set of all superclasses of a class must have a maximum element?

7.9 How are subclasses populated?

There are several possibilities:

- A subclass can be populated by creating elements with an appropriate constructor. These elements also appear as elements of its superclasses, because of the semantics of the subclass relation.
- A subclass can be populated by moving objects from a superclass into the subclass. So objects can change the most specific type to which they belong during their life. For instance, a person can become a student, then an employee, and finally just a person again.
- A subclass can be populated automatically if it can be defined as containing all the elements of the superclass which satisfy a certain condition.

7.10 Can objects be removed from classes and subclasses? Can objects be deleted?

When an object is removed from a class, because of the semantics of the subclass relation, it is removed also from its subclasses; but when it is removed from a subclass it might be left in the superclasses. Does an operator exist for the explicit deletion of an object, or is this task left to the garbage collector? If an object can be explicitly deleted, how are references to it managed? Has the deletion of an object an effect on its components?

7.11 Are subclasses of the same superclass disjoint? Is the union of the sets of the elements of the subclasses equal to the set of the elements of the superclass?

8 Database Issues

8.1 Is persistence orthogonal to the type system?

Persistence is sometimes a characteristic of a specific type constructor, in which case it is a type issue. Orthogonality of persistence and the type system means that any value can be made persistent, in which case it is not a type issue.

8.2 What kinds of database schema evolution are supported?

Schema evolution means the possibility of making changes to the database schema after the database has been populated. To fully understand this important feature of database programming languages it is useful to distinguish changes which have no effects on application programs and existing data, changes which require only the recompilation of application programs, and changes which require modifications to existing programs and data.

9 Other issues

9.1 To what extent is type inference employed in the language?

9.2 Is there any theory supporting the type system?

Does, for example, the set of type definitions build a theory of the application?

9.3 Are there implementation factors that are essential to the understanding of the system?

For example, do the binding mechanisms provide the necessary semantics for fully understanding the system?

9.4 Are there issues that are not covered in the previous questions that are important to understanding the type system?

9.5 What is the status of the implementation of the system?

10 Conclusions

A framework for the comparison of the type systems in database programming languages has been presented. We have avoided being normative, to allow for the greatest flexibility in the languages being compared. Similarly, we have mostly avoided definitions which may over constrain the utility of the framework. We hope that our framework is comprehensive, but realise that this is unlikely given that the study of these issues has only recently begun. Consequently, we have attempted to create an open-ended framework and we invite others to use the framework to describe their languages, extending and refining it where necessary.

11 Acknowledgements

The authors would like to acknowledge the incisive comments of Malcolm Atkinson, Peter Buneman and Rick Hull in the preparation of this document.

This work was partially supported by Ministero della Pubblica Istruzione, Italy, the Science and Engineering Research Council, U.K. grant number GR/E 75395, NSF IRI 8606424 and the Office of Naval Research, University Research Initiative contract N00014 86-K-0764.

12 References

[CW85] Cardelli L. and Wegner P., "On Understanding Types, Data Abstraction and Polymorphism", *A.C.M. Computing Surveys*, Vol. 17, No. 4 (December 1985), pp.471-522.

[OO88] Proc. 2nd International Workshop on Object-Oriented Database Systems, West Germany. *Lecture Notes in Computer Science, 334.* Springer-Verlag, (September 1988).

For detailed expositions of the issues discussed in this paper, the reader is referred to the following:

Atkinson M.P. & Buneman O.P., "Types and Persistence in Database Programming Languages", *A.C.M. Computing Surveys*, Vol 19, No 2 (June 1987), pp.105-190.

Banchilion F. & Buneman P. (Editors) **Database Programming Languages**. MIT Press. Cambridge Mass 1989.

Atkinson M.P., Buneman O.P. & Morrison R. (Editors) **Data Types and Persistence**. Topics in Information Systems Series, Springer-Verlag. Berlin 1988.

Brodie M.L., Mylopoulos J. & Schmidt J.W. (Editors) **On Conceptual Modelling**. Topics in Information Systems Series, Springer-Verlag. Berlin 1984.

Danforth S. & Tomlinson C. "Type Theories and Object-Oriented Programming", *A.C.M. Computing Surveys*, Vol 20, No 1 (March 1988), pp.29-72.

Dittrich K.R. (Editor) **Advances in Object-Oriented Database Systems**. Lecture Notes in Computer Science, Vol 334, Springer-Verlag, Berlin 1988.

Shriver B. & Wegner P. (Editors) **Research Directions in Object-Oriented Programming**. Mit Press, Cambridge Mass. 1987.