Typing in Object Oriented Database Systems

Andrej Brodnik and Hemin Xiao

Abstract

This paper gives an extensive overview of the problem of typing in general, typing in programming languages, typing in database systems, and specifically typing in object oriented database systems. The survey starts with the definition of type elaborated from a mathematical point of view and from the point of view of an object oriented class hierarchy. There follows a short description of possible methods for typing in database systems. Combining all these we conclude with examples of typing in object oriented database systems.

Contents

1 Introduction **29**
1.1 Why typing . . . 29
1.2 Polymorphism . . . 29

2 Types **30**
2.1 What is a type . . . 30
2.2 Two ways of typing . . . 31

3 Mathematical background **31**
3.1 Power types . . . 31
3.2 Type constructors . . . 32
3.3 Type inference . . . 32
3.4 Type calculus . . . 33
3.5 Operations on records . . . 34
3.6 Quantification . . . 34

4 Natural background **34**
4.1 Classes . . . 35
4.2 Recursive types . . . 35
4.3 Inheritance . . . 35
4.4 Polymorphism . . . 36

5 Typing in databases **36**
5.1 Data in databases . . . 37
5.2 Retrieved data . . . 37

6 Typing in OODB **38**
6.1 Types and OODBS . . . 38
6.2 Polymorphism in OODBS . . . 38
6.3 Subtyping in OODBS . . . 39
6.4 Type evolution in OODBS . . . 40

7 Conclusion **40**

1 Introduction

1.1 Why typing

In the programming language paradigm, a type consists of a possibly infinite set of values and also a set of operations on those values. A type and its use are separated by a type definition and a variable declaration. A variable of a specific type can only have values of that specific type. A type may be atomic or structured. An atomic data type, sometimes referred to as simple or basic data type, is a type that has only nondecomposable values, like an integer or a character. A structured type, sometimes also called composite data type or data structure, is a type whose values are composed of component elements that are related by some structure. This structure must be considered by the user when he wants to access the data structure. Since a data structure is a type, it has a set of operations both on its values and its component elements [SW85]. It is built using already existing data types and type constructors on them like array, set, list and record.

Types are essentially tools to increase correctness and efficiency of programs. By forcing the user to declare the structure of the used objects, the system is able to check for invalid operations. Thus, types are used at compile time to check the correctness of the programs, eliminating the need for run time checks. Another advantage is that procedures can be bound to the call the compile time. Thus, expensive operations such as context creation and procedure calling can be replaced with inexpensive operations such as branching. Moreover, without a type system it is impossible for a compiler to optimize code. Since the compiler has no way of determining the procedure implementing an operation, it cannot find out whether the operation has side effects or affects control flow. Thus, common subexpression elimination, dead code removal, and other standard optimizations are not general applicable.

There are two important concepts about typing: static typing and and strong typing [Weg89]. A programming language is said to be statically typed if the type of all expressions can be determined at compile-time (complete type checking at compile-time). If, at compile-time, the type compatibility of all expressions representing values can be determined from the static program representation, the language is said to be strongly typed (may use some run-time checking). Every statically typed language is always strongly typed, but the converse is not necessarily true since a strongly typed language allows the programmer to use generic operations, such as sorting, that capture the structure of an algorithm which is uniformly applicable to a range of types.

1.2 Polymorphism

Typing in most programming languages is largely monomorphic in that values are deemed to have a single type. This is very restrictive, and polymorphism can be used to provide more flexible typing. Polymorphism is defined to be the ability for a value to have more than one type. The values can therefore be used in several contexts (parameter passing or assignment) with different types.

Polymorphism can be classified as the following [CW85, Rou90, Tha90, FP91, CF91]:

Universal polymorphism – a function works uniformly on an infinite number of types which normally exhibit some common structure in a disciplined manner. Universal polymorphism can be further divided into two subcategories:

Parametric polymorphism: a function has an implicit or explicit type parameter which determines the type of the argument for each application of the function

Inclusion polymorphism: a function works on a range of types which is determined by subtyping relationship. A function defined on a particular type can also operate on any of its subtypes.

Ad hoc polymorphism – a function works or appears to work on several different types which may not exhibit a common structure and may behave in unrelated ways for each type. Ad hoc polymorphism can also be further divided into two subcategories:

Overloading polymorphism: the same variable name is used to denote different functions and the context is used to decide which function is denoted by a particular instance of the name.

Coercion polymorphism: a semantic operation needed to convert an argument to the type expected by a function is provided, in a situation that would be otherwise result in a type error [Mit84, Mar91].

2 Types

2.1 What is a type

Typing is a fundamental concept in computing. We now update our definition of type to emphasize its abstraction. A type is an abstract description in the abstract of a related group of entities [Bla91]. For example, the type integer is commonly understood to denote entities which exhibit properties similar to the mathematical concept of whole numbers.

Abstraction plays a key role in typing. Types have the following three distinctive roles:

Abstraction – Values can be complex entities consisting of a particular structure and an associated set of semantics which could be thought of as being properties of the value. Types provide a shorthand way of signifying a value with appropriate properties. It can be assumed that all values of that type share these properties; they may have other difference, but the properties will remain invariant. Different type systems will specify the properties in different ways.

Composition – Type can be used to provide mechanisms for creating higher level abstraction from existing type abstractions. As we stated in the previous section, composite types are established on top of a set of simple types such as integer and real, character and boolean. Composite types can also be defined using other composite types to implement higher levels of abstraction.

Protection – A type system also provides a level of protection against incorrect or undesirable actions. Type checking ensures that invalid operations are not carried out.

The third role of types is actually type checking. A type error is an action which results or may result in an invalid operation on a value. There are two possible sources for type errors: parameter

passing and assignment. Two approaches for deciding the compatibility of types (also known as type equivalencing) are:

Name equivalencing: two types are equivalent if and only if they have the same name; inflexible but easy to implement.

Structural equivalencing: two types are equivalent if and only if their underlying structure is the same; flexible but difficult to implement.

The most important characteristic of the type checking mechanism is the time at which type errors are detected: static type checking detects type errors at compile time, whereas dynamic type checking detects type errors at run time. Often in a typing system we have both of them, but with an emphasis on one of them.

2.2 Two ways of typing

There are two ways of typing [Car88a]. The first one is common in conventional programming languages such as Pascal and can is derived from standard branches of mathematics. Data is organized as Cartesian products (i.e., record types), disjoint sums (i.e., unions or variant types) and function spaces (i.e., functions and procedures).

The second method can is derived from biology and taxonomy. Data is organized in a hierarchy of classes and subclasses, and data at any level of the hierarchy inherit all the attributes of data higher in the hierarchy. The top level of the hierarchy is usually called the class of all objects; every datum is an object and every datum inherits the basic properties of objects, like the ability to tell whether two objects are the same or not. Functions and procedures are also considered as local actions of objects, as opposed to global operations.

These two different ways of typing have generated distinct classes of typing systems. Typing with taxonomically organized data is often called object-oriented, and has been advocated as an effective way of structuring database and large systems in general.

3 Mathematical background

Cardelli claims that the space of all possible values can be partitioned into different sets. Each set induces a type [CW85]. Mathematics defines numerous operations on sets. Some of them make sense on types as well.

3.1 Power types

In set theory there is well defined notion of *power sets*. Cardelli in [Car88b] introduces the similar notion of *power types* to describe subtyping and inheritance. He uses the definition:

$$\forall A : A \subseteq U \Rightarrow A \subseteq \mathcal{P}(U)$$

Note that all subsets of a given set are also subsets of the power set. This is different from the definition:

$$\forall A : A \subseteq U \Rightarrow A \in \mathcal{P}(U)$$

Using the subset relation somehow flattens the type hierarchy which causes not all expressions to be statically typable [Car88b]. On the other hand use of the member relation leads us to so-called kinds and their hierarchy (see for example [GTL89]). In this hierarchy the values represent the hierarchy at the lowest level. The next level is types and so on. Therefore the power set $\mathcal{P}(U)$ would be a level above the types in a hierarchy.

3.2 Type constructors

The next major question to be answered is how to efficiently describe sets. This description is in fact type definition. Most of the authors use the following type constructors and associated semantics (see [CW85, Car88b, Car88a, Rem89, OB89a] and others):

- basic type

$$\frac{E \vdash a : \tau}{E \vdash a : \tau}$$

- Cartesian product

$$\frac{E \vdash a : \tau_1 \qquad E \vdash b : \tau_2}{E \vdash (a \times b) : (\tau_1 \times \tau_2)}$$

- union

$$\frac{E \vdash a : \tau_1 \qquad E \vdash b : \tau_2}{E \vdash [a, b] : [\tau_1, \tau_2]}$$

- function

$$\frac{E \vdash a : \tau_1 \qquad E \vdash b : \tau_2}{E \vdash fun(a)b : \tau_1 \rightarrow \tau_2}$$

All these constructors are well known in programming languages but the most important for our discussion are Cartesian product and union. Union, also known as a variant type, is used in inheritance definition, while Cartesian product can be generalized to record types. The notion of records introduces additional parts of Cartesian product components called labels. The labels uniquely define different record components. On the other hand the records are natural type descriptions of objects [BO90].

3.3 Type inference

As written initially the goal is to write programs with as little additional notation as possible. Such additional notations are also type descriptions, which we would like to omit and leave to the compiler to deduce them. The procedure of deduction is known as type inference. The most widely used algorithm for type inference is Milner's one, which is also used in the programming language ML [Rea89]. This algorithm with certain adaptations is also used in many works related to our topic (see [Wan87, Wan88, Wan89, JM88] and others).

3.4 Type calculus

We are now ready to proceed with the description of calculi over records and their effect on type inference. The most extensive description of the problem can be found in the works of Cardelli [Car88a], and Cardelli and Mitchell [CM89a, CM89b].

Cardelli in [Car88a] defines two basic operations on types, join and meet. Using these two operations one can express all other operations. The meaning of join ($\uparrow$) and meet ($\downarrow$) on two types is defined as (due to [Car88a]):

join operation

- Simple type
$$\frac{E \vdash \tau}{E \vdash \tau \uparrow \tau = \tau}$$
- Records
$$\frac{E \vdash (\forall i : \tau_i \uparrow \tau'_i \quad \text{are defined}) \qquad (\forall j,k : b_j \neq c_k)}{E \vdash \{a_i : \tau_i, b_j : \sigma_j\} \uparrow \{a_i : \tau'_i, c_k : \rho_k\} = \{a_i : \tau_i \uparrow \tau'_i\}}$$
- Variants
$$\frac{E \vdash (\forall i : \tau_i \uparrow \tau'_i \quad \text{are defined}) \qquad (\forall j,k : b_j \neq c_k)}{E \vdash [a_i : \tau_i, b_j : \sigma_j] \uparrow [a_i : \tau'_i, c_k : \rho_k] = [a_i : \tau_i \uparrow \tau'_i, b_j : \sigma_j, c_k : \rho_k]}$$
- Function
$$\frac{E \vdash \sigma, \sigma', \tau, \tau'}{E \vdash (\sigma \to \tau) \uparrow (\sigma' \to \tau') = (\sigma \downarrow \sigma') \to (\tau \downarrow \tau')}$$

meet operation

- Simple type
$$\frac{E \vdash \tau}{E \vdash \tau \downarrow \tau = \tau}$$
- Records
$$\frac{E \vdash (\forall i : \tau_i \downarrow \tau'_i \quad \text{are defined}) \qquad (\forall j,k : b_j \neq c_k)}{E \vdash \{a_i : \tau_i, b_j : \sigma_j\} \downarrow \{a_i : \tau'_i, c_k : \rho_k\} = \{a_i : \tau_i \downarrow \tau'_i, b_j : \sigma_j, c_k : \rho_k\}}$$
- Variants
$$\frac{E \vdash (\forall i : \tau_i \downarrow \tau'_i \quad \text{are defined}) \qquad (\forall j,k : b_j \neq c_k)}{E \vdash [a_i : \tau_i, b_j : \sigma_j] \downarrow [a_i : \tau'_i, c_k : \rho_k] = [a_i : \tau_i \downarrow \tau'_i]}$$
- Function
$$\frac{E \vdash \sigma, \sigma', \tau, \tau'}{E \vdash (\sigma \to \tau) \downarrow (\sigma' \to \tau') = (\sigma \uparrow \sigma') \to (\tau \uparrow \tau')}$$

The resulting type of a join over records is the biggest common subset of joined types (intersection) and the resulting type of meet is the smallest common superset of operands (union).

Using these two operations and the notion of subtype Cardelli and Mitchell define the transitive and reflexive relation <:, which organizes the space of types as a lattice [Sta88]. In the lattice Stansifer also defines two special types. They are bottom ($\perp$: in [CM89a] the record of all possible labels) and top ($\top$: in [CM89a] $\ll\gg$ denoting an empty record, or in [CW85] Top set).

3.5 Operations on records

It remains to solve the problem of computing meet and join operations. The first one, meet, is especially important in object oriented programming. In an important special case it can be defined as concatenation [Wan89] or through the use of a number of with and without operators [Rem89, Wan87][1]. Both Remy and Wand initially assume that the set of all possible labels in records is finite. Wand [Wan88, Wan89] then defines the resulting type of the meet operation using with and without operators.

On the other hand Remy [Rem89] associates with each label a flag. If the flag is set the label is present in the record and otherwise not. In spite of this simplification and strict notation there remains a problem of typing $(A \parallel B).a$, where $\parallel$ denotes the concatenation of records A and B, and $.a$ is a label of the concatenated record. The problem occurs when $a \in A$, $a \in B$, and $A.a \neq B.a$. Cardelli and Mitchell [CM89a, CM89b] avoided this problem by adding the rename operator.

3.6 Quantification

Another aspect of defining types as sets is quantification. In general quantification can be either universal ($\forall$) or existential ($\exists$), and their usage and definition was introduced by Cardelli and Wegner in [CW85]. They also introduced a bounded quantification and used it in typing the object oriented class hierarchy.

Their usage of existential quantification was limited to the abstraction of data types and information hiding in module interfaces. In their terms the stack module would be defined as:

$$\begin{aligned} \exists Stack, \forall \tau \quad : \quad & \\ & Pop : Stack \to (\tau \times Stack), \\ & Push : (\tau \times Stack) \to Stack, \\ & Full : Stack \to Boolean, \\ & Empty : Stack \to Boolean \end{aligned}$$

Canning et al. [CCHO89] used essentially the same approach, but because they were dealing only with a strict object oriented class hierarchy, they did not need to introduce bounded quantification. They replace it with a parametric polymorphism. The same idea is already used in Simula67 [Sve87].

4 Natural background

The relation between two types in the previous section was computed from their structural information. In this section this information will be given in advance through the so called class hierarchy. Classes have signatures and these signatures are essentially types, which gives us a relation defined in [CM89a]. It defines $A <: B$ if A is a subtype of B.

[1] The operator with describes a record with a given label, while the meaning of without is opposite.

4.1 Classes

The object oriented programming paradigm includes the concept of a so called class hierarchy. Here classes are essentially generic object descriptions, and when they are instantiated objects are born [Sve87].

The classes themselves have a type description called the signature. The objects, when instantiated, have the same signature. In general the signature is a record type as defined in the previous section [Car88a]. The fields in the records are defined by labels and may be of an arbitrary type (basic types, records, variants, functions, ...). If a type of certain field is a function the field is also called a method or message. Typing of such fields is the the most difficult problem.

4.2 Recursive types

Any of the record fields may introduce a recursive type definition (see for example [MPS84, AC91]). It is perfectly legal to define a type as:

$$person = \{child : person\}$$

and the compiler will have to deal with expressions of the form

$$a.child.child \ldots child$$

where $a : person$.

This brings up the problem of structural equivalence checking which is more complicated because the structure is no longer a simple directed graph, but a graph with cycles. The problem is addressed in more detail by MacQueen et al. in [MPS84] or by Amadio and Cardelli in [AC91].

4.3 Inheritance

As previously mentioned, the classes form a hierarchy (which happens to be a lattice [Sta88]). This hierarchy is implicitly defined through subclass and superclass relations. Namely, each class signature is defined to be the same as its superclass with the addition of some fields. Semantically, a subclass inherits from a superclass fields along with typing information. This is how an inheritance is defined (compare for example with [FP88], or with [CHC90] where Cook et al. also show that subtyping is not same as inheritance).

The notion of inheritance can be generalized to multiple inheritance where a subclass can inherit fields from an arbitrary number of superclasses. Note that adding a new field to a class and creating a subclass this way:

$$\text{class } A = \text{subclass } B \text{ with } l : \tau$$

is equivalent to

$$\text{class } C = \{l : \tau\}$$

$$\text{class } A = \text{subclass } B \text{ and } C$$

This form is more general and was essentially already studied in a previous section through the meet and join operators. Further references can be found in Harper and Pierce [HP90, HP91] or Mitchell [Mit88].

A slightly different approach was proposed by Brezau-Tannen et al. in [BTCGS89]. Their suggestion is based on coercion polymorphism and proposes that subclasses do not inherit fields from superclasses, but instead new interfaces are created for them (casting functions).

4.4 Polymorphism

The major polymorphic usage is the usage of superclass methods on subclasses. In general we have typing:

$$fun : \tau_1 \to \tau_2$$

Because of subtyping rules as defined in [CW85] one can use instead of τ_1 any type which is a subtype of τ_1.

Cardelli and Wegner in [CW85] already pointed out that such typing can lead to lose of information. For example if we define:

$$\tau_1 = \{a : \sigma_1\}, \quad \rho = \tau_1 \downarrow \{b : \tau_2\}, \quad f : \tau_1 \to \tau_1$$

and apply a function call which is typed as:

$$f(x : \rho) : \tau_1$$

this is not always what we want because we would like to get back type ρ. Therefore Cardelli and Wegner introduced the notion of bounded quantification. Using it we would type the previous function definition as:

$$\forall \tau_1, \tau_2 : (\tau_2 \subseteq \tau_1) \Rightarrow f : \tau_2 \to \tau_2$$

Later it was shown that the bounded quantification is sometimes difficult to type. Therefore, simpler models were introduced. The one with coercions was already mentioned [BTCGS89]. Further, let us mention F-bounded quantification [CCH$^+$89], and enriched quantification by Harper and Pierce [HP90, HP91]. They allow predicates in bounds of quantification to be more complicated and they can also include the additional operators lacks and has. Their meaning is similar to the meaning of the without and with operators, respectively.

In reality, a compiler type checker uses both natural and mathematical approaches. In the case of object oriented languages, the natural view is used in the declaration part of the compiled program where the class hierarchy is built. The mathematical approach serves its purpose in the typing of expressions used in the programme when structural equivalences are computed.

Both approaches are equally powerful; however, in the object oriented case the checker need not exploit the complete type space because of the class hierarchy.

5 Typing in databases

All that was discussed in previous sections concerning types is also directly applicable to databases. In general, records saved in the same file are of the same type which is usually a record type.

5.1 Data in databases

Variables saved in databases are easily described using record types. However, there is an important simplification because their fields are not functions [OB89b]. Therefore we have two coexistent type systems. One, for non-persistent variables, includes functions, while the other, for persistent variables, does not. The remaining type constructors (union, product, recursion) are used over both variables.

5.2 Retrieved data

In relational databases the basic type is a relation. It is in fact a record type. On this assumption Buneman and Ohori in [BO90] built their type system which is used in the language *Machiavelli*. In their system types of entities are inferred through an *is-a* rule.

Beside the typing of entities, queries have to be typed as well. Therefore they introduce a new type constructor, set, which is used to type the result of a query. Using this construct we can type a query as:

$$\forall\tau : query : (\tau \times (\tau \to Boolean)) \to set(\tau)$$

Note that the resulting set consists of elements of the same type. On the other hand it would be also reasonable to type the query as:

$$\forall\tau, \sigma : \sigma \subseteq \tau : query : (\sigma \times (\sigma \to Boolean)) \to set(\sigma)$$

The authors also noted this problem but did not give any solution for it.

In addition to the type constructor for sets, the authors also provide the basic polymorphic operation over sets called hom. It is defined recursively as:

$$\begin{array}{ll} hom & (f, op, z, \{\}) = z \\ hom & (f, op, z, \{e_1, e_2, \ldots, e_n\}) = \\ & op(f(e_1), op(f(e_2), \ldots, op(f(e_n), z)\ldots)) \end{array}$$

Essentially, the hom operation applies the function f to each element of the set given as the last argument. The results are finally combined using function op with the third argument z as a seed of a new structure. For example if $op = union$, hom would map a set of elements onto another set of elements where function f would be applied to each element.

The described operator is powerful enough to perform all relational algebra operations. Further, because it is simply typed:

$$hom : (\tau_1 \to \tau_2, (\tau_2 \times \tau_3) \to \tau_3, \tau_3, set(\tau_1))$$

all these operations are simply typed as well. But the problem of determining a type equivalency is in general NP-complete [OB89b].

Buneman and Ohori also give a specialization of their general typing system for object oriented databases. In this specialization they use the assumption that objects are related to record types through the *is-a* relation, and the polymorphism from the typing system [OB89b, BO90].

6 Typing in OODB

6.1 Types and OODBS

In the object-oriented database system (OODBS) paradigm, a type may be defined by a predicate for recognizing expressions of the type. A class can be seen as a special kind of type, namely a type whose predicate is a template specification. For every class there is a type predicate that characterizes the set of all potential instances of the class, namely the predicate "is an instance of the class" [WZ88].

Such a definition leaves some leeway for interpretation. It does not require that the structures of all instances of a class are identical, as it is usually required in structural object-oriented systems. Since a class commonly realizes the principle of data abstraction, its instances hide their internal representation. This means that the type specification of a class does not necessarily say anything about the concrete structure of its instances. In other words, a type may be seen as a set of values which belong to a specific category without, at the same time, determining the structure of the values. The question of whether an object is of a specific type is determined by its functionality and behavior and not by its structure. This understanding of the notion of type is quite common in object-oriented systems. In many object-oriented systems the internal representation of instances of a class is defined by a fixed number of instance variables which are not bound to a specified type. That is, the structure of an instance of a class is only described by the number of its instance variables while the structure (data type) of the instance variables is not laid down. In such an environment it is the object, not the variable, that is typed [Tho89].

An instance of a class is created by assigning instances of (other) classes to its instance variables. Due to the principle of data abstraction, the value of an instance variable can be accessed by the operations which are associated with that value. This not only allows the user to collect instances of different structures in one class but it also permits an extensive use of inheritance. However, it requires the concept of late binding to be realized. While many people argue that such an object world, which is based on "run-time-typing", is an essential feature for a programming language to be object-oriented as it provides more flexibility, others are convinced that the price to be paid (less efficiency and less run-time safety) is too high, therefore, demanding some form of early typing (static or strong typing).

We will call a class typed if its attributes are typed. This is to say that the system provides a kind of static or strong typing (for example, C++) and untyped otherwise (for example, SMALLTALK) [US90]. According to this definition, two kinds of object-oriented database systems can be distinguished. One category of systems supports typed classes and the other category of systems does not. Object-oriented database systems like O_2 [VBD89], OOPS [US89], OZ+ [WL89], and VBASE [AH87] belong to the first category while Gemston [MSOP86], VISION [CS87], and ORION [BC$^+$87] are examples of the second.

6.2 Polymorphism in OODBS

In object-oriented systems we are mainly concerned with inclusion polymorphism, overloading, or parametric polymorphism.

If we want to fully exploit the advantages of inheritance and polymorphism we need some kind of dynamic type checking and late binding. In general, late binding makes type checking more

complicated and in some cases impossible, but it still permits a good deal of compile-time type checking.

The developers of VBASE recognized that 90% of all type checking and method binding can be done at compile-time [AH87]. Johnson thinks that it is even possible to build systems with the flexibility of SMALLTALK or Lisp and with the safety of strong typing [MJ+89]. On the other hand, Danforth and Tomlinson want to keep the advantages of a type-free world [DT88], while at the same time offering the safety and efficiency of early typing. They hold the opinion that flexibility in system description is closely related to the ability to control the time at which components are bound and the strength of such bindings. Therefore, they suggest the incorporation of an explicit typing system that can support but does not require static typing. For instance, Gemstone [MSOP86] follows this approach by offering the facility (but not the need) to constrain the value of an attribute to a specific type [Ame87, AB87, BW86, CW85, HO87, Str87].

Dynamic type checking is precisely the policy adopted in most object-oriented database systems. Type errors are notified by the "method unknown" message.

6.3 Subtyping in OODBS

In OODBMS's, there is often a unique mechanism providing subtyping and inheritance functions. However, inheritance is not subtyping [CHC90]. The difference between the two concepts can be roughly characterized as follows. Inheritance is a reusability mechanism allowing a class to be defined from another class, possibly by extending and/or modifying the superclass definition. On the other hand, a type τ is a subtype of τ' if an instance of τ can be used in place of an instance of τ'. Therefore, subtyping is characterized by a set of rules ensuring that no type violations occur when the instance of a subtype τ replaces an instance of a supertype τ'.

The fact that a class C is a subclass of a class C' does not necessarily imply that C is also a subtype of C'. Subtyping, however, influences inheritance since it can restrict the overriding and can impose conditions on multiple inheritance so that the subtyping rules are not violated. An example of restriction on overriding is the requirement that when the domain of an attribute is defined in a subclass, this domain must be a subclass of the domain associated with the attribute in the superclass [Weg87].

Subtyping can be divided into two groups:

Behavioral subtyping A type τ is a subtype of τ' if τ provides methods with the same name and the same (or compatible) arguments as τ' (and possibly additional ones). The criteria for behavioral subtyping are often based on the notion of conformity [BHJ+87]. Conformity can only be used when the method signatures include the types of arguments and the result.

Structural subtyping (inclusion) A type τ is a subtype of τ' if τ provides the same attributes as τ' or attributes compatible with those of τ' (and possibly additional ones).

Usually, most OODBSs enforce only structural subtyping, even if subclasses inherit both attributes and methods from the superclasses. For example, the O_2 system [DAB+90] uses structural subtyping, while methods use a condition different from conformity. This condition leads to a less restricted type system that does not guarantee that one instance of a subclass can always be safely used in place of another instance of a subclass. In contrast, the Vbase system [AH87] enforces both structural and behavioral subtyping, using the notion of conformity.

6.4 Type evolution in OODBS

In OODBS involved in design environment, a given object will evolve and change state over time. Each object in the database was created as an instance of some type at some point in that type's evolution. The type described all the assumptions about the object's behavior. When the type definition changes, old objects may be incompatible with a new definition of their type.

Skarra and Zdonik [SZ90] address this problem of maintaining consistency between a set of persistent objects and a set of type definitions that can change in OODBS. In order to make a type's change transparent with regard to programs that use the type, they proposed the use of a version control mechanism and a set of error handlers associated with the version of a type.

In their opinion, every object is an instance of some type which describes the behavior of its instance. A type τ is a specification of behavior such as operations which can be applied to τ, properties which are defined for τ, and constraints which must be satisfied by τ. Each type has an implementation that is hidden. Types can be related to each other by means of a special property called *is-a* which induces an inheritance relationship between types. Their system enables multiple inheritance and the set of *is-a* relationships is represented as a directed acyclic graph with exactly one node containing no outgoing arcs (i.e., *type lattice*). The constraint defined on a type delimits the range of acceptable values for properties and operations. Using what they call a constraint language, it can be verified that a newly defined type is a legal subtype of type(s) declared as its supertype(s). Thus, this is the way in which persistent shared objects behavior can be governed.

The problem of changing types arises from the strong notion of typing that is present in most OODBS. There are several other ways of handling it, making this a very interesting topic (see [ABHS84, BKKK86, Bor85b, Bor85a]).

7 Conclusion

In this paper we have presented a survey of typing systems. The survey was on one side general and on the other side tried to emphasize the role of the type constructor record.

Types are very important tools for checking program correctness. Further, the notion and usage of polymorphism enhances code reusability. The third essential tool used in software engineering is type abstraction which, as with the previous two tools, is fully supported in the object oriented paradigm.

It appears that computer science provides a large amount of knowledge and tools to use the object oriented approach in the database world. On the other hand there we have a feeling that there exists a major gap in understanding between the database world and the world of programming languages (compare also with [Nel91]). Namely, it is often true that the same terms used on both sides do not share a common meaning. We believe that more work should be done in order to improve mutual understanding between the two sides. This would make it possible to combine efforts from both sides which would lead to a substantial knowledge upgrade.

References

[AB87] M. Atkinson and P. Buneman. Types and persistence in database programming lan-

guages. *ACM Computing Surveys*, 19(2), 1987.

[ABHS84] M. Ahlsen, A. Bjornerstedt, C. Hulten, and L. Soderlund. Making type changes transparent. Technical Report SYSLAB – 22, University of Stockholm, February 1984.

[AC91] R.M. Amadio and L. Cardelli. Subtyping recursive types. In *ACM Symposium on Principles of Programming Languages*, 1991.

[AH87] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1987.

[Ame87] P. America. Inheritance and subtyping in a parallel object-oriented programming language. In *European Conference on Object-Oriented Programming*, 1987.

[BC+87] J. Banerjee, H.T. Chou, et al. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1), Jan 1987.

[BHJ+87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Trans. Software Eng.*, SE-13(1):65–76, 1987.

[BKKK86] J. Banerjee, H.J. Kim, W. Kim, and H.F. Korth. Schema evolution in object-oriented persitent databases. In *Proceedings of the sixth Advance Database Symposium*, August 1986.

[Bla91] G.S. Blair. *Object-Oriented Languages, Systems and Applications*, chapter Types, Abstract Data Types and Polymorphism, pages 75–107. Pitman, 1991.

[BO90] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. Technical Report MS-CIS-90-64, University of Pennsylvania, 1990.

[Bor85a] A. Borgida. Features of languages for the development of information systems at the conceptual level. *IEEE Software*, 1(2):63–72, January 1985.

[Bor85b] A. Borgida. Language features for flexible handling of exceptions in information systems. *ACM Transactions on Database Systems*, 10(4), December 1985.

[BTCGS89] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrow. Inheritance and explicit coercion. In *IEEE Symposium on Logic in Computer Science*, 1989.

[BW86] K. Bruce and P. Wehner. An algebraic model for subtypes in object-oriented languages. *SIGPLAN Notices*, 21(10), 1986.

[Car88a] L. Cardelli. The semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Car88b] L. Cardelli. Structural subtyping and the notion of power types. In *ACM Symposium on Principles of Programming Languages*, 1988.

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Conference on Functional Programming Languages*, 1989.

[CCHO89] P.S. Canning, W.R. Cook, W.L. Hill, and W.G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1989.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *ACM Conference on Programming Language Design and Implementation*, Toronto, 1991.

[CHC90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. In *ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[CM89a] L. Cardelli and J.C. Mitchell. Operations on records. Technical Report Research report 48, Digital Equipment Corporation, System Research Center, August 1989.

[CM89b] L. Cardelli and J.C. Mitchell. Operations on records – summary. In *Conference on Mathematical Foundations of Programming Languages Semantics*, volume 442, pages 22–52, Lecture Notes in Computer Science, 1989.

[CS87] M. Caruso and E. Sciore. The vision object-oriented database system. In *Proc. of Workshop on Database Programming Languages*, 1987.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[DAB+90] O. Deux, G. Arango, G. Barbedette, V. Benzaken, G. Bernard, et al. The story of O_2. *IEEE Trans. Knowledge and Data Eng.*, 2(1):91–108, 1990.

[DT88] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1), 1988.

[FP88] Y.C. Fuh and P.Mishra. Type inference with subtypes. In *European Symposium on Programming*, 1988.

[FP91] T. Freeman and F. Pfenning. Refinement types for ML. In *ACM Conference on Programming Language Design and Implementation*, Toronto, 1991.

[GTL89] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, Cambridge, 1989.

[HO87] D. Halbert and P. O'Brien. Using types and inheritance in object-oriented programming. In *European Conference on Object-Oriented Programming*, 1987.

[HP90] R.W. Harper and B.C. Pierce. Extensible records without subsumption. Technical Report CMU-CS-90-102, Carnegie Mellon University, School of Computer Science, Pittsburgh, February 1990.

[HP91] R. Harper and B. Pierce. A record calculus with symmetric concatenation. In *ACM Symposium on Principles of Programming Languages*, 1991.

[JM88] L.A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Lisp and Functional Programming*, 1988.

[Mar91] J.A. Mariani. *Object-Oriented Languages, Systems and Applications*, chapter Object-Oriented Database Systems, pages 166–195. Pitman, 1991.

[Mit84] J.C. Mitchell. Coercion and type inference. In *ACM Symposium on Principles of Programming Languages*, 1984.

[Mit88] J.C. Mitchell. Polymorphic type inference and type containment. *Information and Computation*, 76:211–249, 1988.

[MJ+89] E. Moss, R. Johnson, et al. Inheritance: Can we have our cake and eat it too? In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1989.

[MPS84] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *ACM Symposium on Principles of Programming Languages*, 1984.

[MSOP86] D. Marier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented dbms. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1986.

[Nel91] M.L. Nelson. An object oriented tower of babel. *OOPS Messenger*, 2(3):3–11, July 1991.

[OB89a] A. Ohori and P. Buneman. Static type inference for parametric classes. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1989.

[OB89b] A. Ohori and P. Buneman. Type inference in a database programming language. In *ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, July 1989.

[Rea89] C. Reade. *Elements of Functional Programming*. Addison-Wesley, Reading, Massachussets, 1989.

[Rem89] D. Remy. Typechecking records and variants in a natural extension of ML. In *ACM Symposium on Principles of Programming Languages*, 1989.

[Rou90] F. Rouaix. Safe run-time overloading. In *ACM Symposium on Principles of Programming Languages*, 1990.

[Sta88] R. Stansifer. Type inference with subtypes. In *ACM Symposium on Principles of Programming Languages*, San Diego, California, 1988.

[Str87] B. Stroustrup. What is "object-oriented programming?". In *European Conference on Object-Oriented Programming*, 1987.

[Sve87] Svensk standard SS 63 61 14. Databehandling programspråk – SIMULA, 1987.

[SW85] D. Stubbs and N. Webre. *Data structures with abstruct data types and Pascal*. Brooks/Cole Publishing Company, 1985.

[SZ90] A.H. Skarra and S.B. Zdonik. *Research Foundations in Object-Oriented and Semantic Database Systems*, chapter Type evolution in an object-oriented database, pages 137–155. Prentice Hall, 1990.

[Tha90] S.R. Thatte. Quasi-static typing. In *ACM Symposium on Principles of Programming Languages*, 1990.

[Tho89] D. Thomas. What's in an object? *BYTE*, March 1989.

[US89] R. Unland and G. Schlageter. An object-oriented programming environment for advanced database applications. *Journal of Object-Oriented Programming; SIGS Publications*, 2(1), May/June 1989.

[US90] R. Unland and G. Schlageter. Object-oriented database systems: Concepts and perspectives. In *Database Systems of the 90s*, 1990.

[VBD89] F. Veley, G. Bernard, and V. Darnis. The o_2 object manager: an overview. In *International Conference on Very Large Data Base*, 1989.

[Wan87] M. Wand. Complete type inference for simple objects. In *IEEE Symposium on Logic in Computer Science*, 1987.

[Wan88] M. Wand. Corrigendum: Complete type inference for simple objects. In *IEEE Symposium on Logic in Computer Science*, 1988.

[Wan89] M. Wand. Type inference for record concatenation and multiple inheritance. In *IEEE Symposium on Logic in Computer Science*, 1989.

[Weg87] P. Wegner. *Research Directions in Object-Oriented Programming*, chapter The Object-Oriented Classificatio Paradigm, pages 479–560. MIT Press, 1987.

[Weg89] P. Wegner. Learning the language. *BYTE*, March 1989.

[WL89] S. Weiser and F. Lochovsky. *Object-Oriented Concepts, Databases, and Applications*, chapter An Object-Oriented Database System. Addison-Wesley Publishing Company, 1989.

[WZ88] P. Wegner and S. Zdonik. Inheritance as an incremental modification mechanism or /what like is and isn't like. In *European Conference on Object-Oriented Programming*, 1988.