December 1993

A Cardinal Typing Approach for An Integrated Object Modeling

William Chao
National Sun Yet-Sen University

A CARDINAL TYPING APPROACH FOR AN INTEGRATED OBJECT MODELING

William S. Chao

Information Management Department
National Sun Yat-Sen University, Taiwan 80424, R.O.C.

ABSTRACT

Object-oriented technology, which includes object-oriented analysis, design, and implementation, has come out as an eminent practice for large system development. Using object-oriented technologies would save us a lot of implementation efforts and improve the maintainability of the whole system. However, conventional object-oriented technologies, such as C++ and Smalltalk are inefficient in utilizing a good typing mechanism. Therefore, some deficiencies occur during the object-oriented modeling. For example, generic functions are treated very differently from parametric polymorphisms in most cases. There are many more such dis-integration phenomena if we carefully examine them in details. This makes their modeling more complex and then difficult to be synthesized. In this paper, we present a model, abbreviated TOM for The Object Modeling, which completely follows the framework of a nice type theory practice. By doing so, TOM is able to integrate (1) the database manipulation language with the host language, (2) parametric polymorphisms with subtyping ones, (3) universal polymorphisms with overloading ones, and (4) generic functions with inheritance.

1. INTRODUCTION

Object-oriented technology has come out as an important practice for large system development. Part of the reason is because object-oriented method uses "modulation" and "encapsulation" mechanisms to hide implementation details from users; the other part of the reason is it lets programmers to reuse their designs as well as implementations by applying the "inheritance" technique. Using object-oriented technologies would reduce much development efforts and improve the maintainability of the whole system once it has been completed

In the following sections, we will first introduce what the object-oriented methodology is. Most of those popular terms, such as modulation, encapsulation, and inheritance are briefly discussed in Section 2. In Section 3, the typing mechanism for object orientation is elaborated there. Section 3, together with Section 4, construct the most essential part of The Object Modeling (TOM.) Also, presented in Section 4 are the syntax as well as the semantics of TOM.

The major advantage of having TOM is in its ability to integrate (1) the database manipulation language with the host language, (2) parametric polymorphisms with subtyping ones, (3) universal polymorphisms with overloading ones, and (4) generic functions with inheritance. These characteristics are then discussed in Section 5. Finally, we will have a conclusion of this paper in Section 6. Some possible future researches will also be included in this section.

2. OBJECT ORIENTED METHODOLOGY

An object is an instantiation of a class and a class is an abstract data type [Lisk86], which consists of (a) a data structure definition for the class, and (2) a collection of operations, called "methods," that apply to the objects in the class. In other words, a class equals to the data structure definition plus methods within that data structure.

Some good definitions of object-oriented terminology can be found in ([Cox86], [Gold89].) Readers are encouraged to take a look at them. Here we only briefly mention the most key concepts of the object oriented technology. They are (1) modulation, (2) encapsulation, and (3) inheritance. Let us start from modulation.

2.1. Modulation

Modulation [Dahl72] is an important feature in object-oriented approach. The term "modulation" has been used for a long time since Dijkstra first tried to promote the most famous structured programming concept.

In general, modulation demands us to divide an algorithm into modules. Making it so has one advantage of localizing the problems whenever they appear. This helps us to improve the maintainability once a system is built up.

2.2. Encapsulation

If we call modulation is a mechanism of structuring the algorithm, then encapsulation is the other one which is used to structure the data. The concept of encapsulation first came from the abstract data type, abbreviated ADT [Lisk80]. As modulation, encapsulation mechanism is completely adopted by the object-oriented technology.

Encapsulating functions into the data structure which can only be accessed through these encapsulated functions assists us to have a very safe control of those data. Besides this, encapsulation mechanism also provides us the ability to hide the implementation details from users.

2.3. Inheritance

The basic idea behind inheritance [Meye88] is as follows. If the type of data structure "B" is a subtype of the type of data structure "A" and suppose we have already had a method "M" defined for "A," then there is no need to re-define another method "M" for "B." Data structure "B" can just use this method "M," directly.

Inheritance implies software reusability [Booc87]. As only as there is a subtype relationship between two data structures, there are methods which can be inherited. Our object modeling will fully utilize this capability.

3. TYPING MECHANISM FOR OBJECT ORIENTATION

The term "type" is used to represent a collection of values. Recall from Section 2.3. that inheritance mechanism is achieved completely through the subtyping relationship among data structures. Therefore, it is necessary for us to elaborate the needed type theory which forms the basic background of modeling the object-oriented technology.

3.1. Cardinal Typing System

Primitive types are basic types which construct the fundamental collection of values. For example, type "integer" is a collection of all integer numbers while type "Boolean" stands for the aggregation of two values, i.e. true and false. Primitive types are sometimes called primitive domains, (e. g., see [Chao88].)

It is ideal to represent types in a cardinal way. For example, (integer, real, Boolean) stands for a type which has three cardinalities. The first one is an integer type, the second one is a real type, while the third one is a Boolean type.

In the cardinal typing system, the cardinal position plays an important role. For example, (integer, real) is different from (real, integer). The first cardinal position of (integer, real) is an integer type while the first cardinal position of (real, integer) is a real type.

The cardinality number can only be increased through the Cartesian product (i. e., constructor $\times$) operation which will be discussed in the Type Construction Section.

3.2. Type Construction

In addition to the primitive types of integer, Boolean values, etc., there may be also structured types. There are several ways of constructing new types from simpler types. In general, four constructors which operates on type primitives. They are (1) $+$, (2) $\times$, (3) $\rightarrow$, and (4) $\downarrow$.

Constructor "$+$" is mainly used to add two primitive types into one. For example, A $+$ B produces a type which is a supertype of both A and B. Constructor "$\times$" is sometimes called Cartesian Product which has the contrary effect of constructor "$+$." In general, A $\times$ B generates a type which is a subtype of both A and B. Besides this, constructor $\times$ also has the effect of increasing the cardinality number in the cardinal typing system. For example, if types "A" and "B" have m and n cardinalities, respectively, then the resulting cardinality number of A $\times$ B will be m+n. The third one, "$\rightarrow$," makes a function type. Usually, A $\rightarrow$ B means that it is a type of a function which has the mapping from type A to type B. The last one, "$\downarrow$," is called a subtyping constructor. Let us also see an example here. A $\downarrow$ B means we derive a new type B which is a subtype of type A.

Through type operation, a type expression may have arbitrarily complex structure. Expressions for type contain several constructors, with "↓" taking precedence over "×," and "×" taking precedence over "+," and "+" taking precedence over "→." The type operator "→" associates to the right. Thus A → B + C × D → E ↓ F means A → ((B + (C × D)) → (E ↓ F))

An implementation of the cardinal type system together with type construction will be shown in another paper [Chao93].

3.3. Generic Type

There is a special type which is, in most cases, called a generic type. We use "*" to stand for a generic type. A generic type is sometimes called a parameterized type [Stro88]. Within the same cardinal position, a generic type is always a supertype of any other types, e. g., integer, real, Boolean, string, etc. If we insist using a formula to represent the generic type, then the following one is a nice try.

* = integer + real + Boolean + string + ...

The generic type plays an important role in our object-oriented modeling. It can be used to define the universal type (as we shall see it immediately following this section.)

Besides this, the generic type is also useful in defining types of generic functions. This is achieved through the operation of constructor "→." For example, * → * stands for the functional type which is a mapping from a generic type to another generic type. If we instantiate type "*" with "integer," then this functional type will become "integer → integer" which stands for a mapping from an integer to an integer. If we prefer those two generic types to be instantiated differently, then instead of using "* → *" we shall use * → **. In this case, "*" and "**" all represent the generic type. However, they can be instantiated differently. One possible situation is to get a type which is integer → real. That is the first generic type is instantiated to be integer and the second one is instantiated to be a real type.

For the "+" constructor operating on a generic type, we have the following theorem.

Axiom 3.3.1. * = * + A = A + *,
for A to be any kind of types.

For the "×" constructor operating on a generic type, we have the following theorem.

Axiom 3.3.2. A = A × *
for A to be any kind of types.

Let us spend a little effort to explain the meaning of the above axiom. Suppose type "A" has n cardinality already. Then for those undefined cardinality, i. e., n+1, n+2, etc. we can replace each of them with a generic type.

For the "↓" constructor operating on a generic type, we have the following theorem.

Axiom 3.3.3. A = * ↓ A
for A to be any kind of types.

3.4. Universal Type

The universal type is a supertype of any kind of types. It can be defined as:

Definition 3.4.1. Universal type = *

From the above definition, we see that the universal type is a stand-alone generic type.

Axiom 3.4.1. * = *× * = *× *× * =

Axiom 3.4.1. tells us that if the type has more than one cardinalities, then for each cardinal position to be a generic type in order to make this type an universal type.

4. THE OBJECT MODELING

The Object Modeling, abbreviated TOM, heavily depends on the framework of the cardinal typing and type construction mechanisms discussed in the previous section. In this Section, we will show how TOM works and used to model the real world objects and classes.

4.1. Class Definition

In TOM, a class is defined as

```
class_name:     R2
super_class:    R1
attributes:     type1   a1;
                type2   a2;
                type3   a3;
                .....
methods:        type4   m1 (a1, a3) body1
                type5   m2 (a2) body2
                type6   m3 (a1, a2, a3) body3
                .....
```

From the above definition we see that "R2" is the name of this class. Super_class denotes that "R1" is the super class of "R2". There are many attributes for class "R2". In this case, types of attribute "a1," "a2," and "a3" are "type1," "type2," and "type3," respectively. "Type1," "type2," and "type3" actually are type expressions which will be discussed in more detail when we go through Section 4.3. Also shown above are methods of class "R2." Method "m1" which has "a1" and "a3" as its input parameters. Note that "a1" and "a3" are attributes of "R2". The body of method "m1" is defined in "body1". For method "m2," it has function body "body2" and one input parameter "a2." As the return type of this method, "type5" is it. A similar explanation as method "m1" and "m2" applies to method "m3".

4.2. Two Basic Classes

In our object modeling, there are two basic classes: records and tables. Records are more or less like "user objects" while tables are sometimes cailed "database objects" [Ullm88]. In general, TOM allows arbitrary code in methods but distinguishes between record objects and table objects. Let us find out more about these two classes.

4.2.1. Record Classes

Record class is the most general class, just like the Object Class in Smalltalk language [Gold89]. To define a new record class, we need to give its superclass, attributes, and methods as well. A record class definition is something like:

```
class_name:     R3
super_class:    Record
attributes:     type7   a4
                type8   a5
                .....
methods:        type7   m4
                type7   m5
                .....
```

In the above definition, we see that class "R3" is a record class and it has two attributes, i. e., "a4" and "a5" which belong to the type "type7" and "type8," correspondingly.

A subclass can be derived from a defined superclass. For example, if "R9" is a subtype of "R5" and "R5" is defined as a record class then "R9" is a subtype of a record class, automatically.

4.2.2. Table Classes

Table classes are similar to the collection classes in the Smalltalk language [Gold89]. A table is a collection of objects of some record classes. The main purpose for TOM to have this class is to support the database objects. A general table class would look like:

```
class_name:     T1
super_class:    Table
attributes:     type9   a6
                type10  a7
                .....
methods:        type11  m6
                type12  m7
                .....
```

The only difference between a record class and a table class is that the latter one may have zero to many records while the former one always has only one record. There are kinds of methods which may only operates on the relationships among records. Therefore, these methods can only exist in the table classes. On the other hand, we do not have any methods of this kind operation for a record class.

4.3. Type Expressions in TOM

As we see in the previous sections that type expressions play an important role in the definition of a class. They occur, in general, either in the type definition of attributes or in return type definitions of a method.

To reduce complexities, only two kinds of constructors, "+" and "↓" are allowed in the TOM's type expressin. A typical type expression looks like A + B ↓ C, where A, B, and C are primitive types. This may be the most complicated type expressions for an attribute definition in TOM. In most cases, an attribute definition is nothing more than a simple primitive type which is a subtype of the generic type within the same cardinal position.

It is also possible that the type of an attribute is a generic type. In this case, it would be very useful when a generic function, which will be discussed in later section, is defined.

In the case of reusing the parametric polymorphism, we may need the following expression to represent a primitive type.

(R1.a1)↓integer

In the above expression, "R1" stands for a class and "a1" is an attribute of "R1." "(R1.a1)↓integer" means to subtype the type of "a1" to an integer type within the same cardinal position where attribute "a1" stands. The usage of this kind of expression will be discussed more in Section 5.2.

4.4. Typing Mechanism of Attributes

The effect of adding an attribute is more or less like doing Cartesian product, i. e., constructor ×, on a type. For example, if we define a new class "RR2" which has class "RR1" as its supertype and "aa1," and "aa2" as its attributes. Then the type of the data structure of class "RR2" will be type(aa1) × type(aa2) × type(aa3) × type(aa4)., supposing "RR1" has "aa1" and "aa2" as its attributes. Note that "aa1" and "aa2" are not explicitly dfined in "RR2." Instead, they are inherited from "RR1" which is the superclass of "RR1."

As we have already pointed out in the previous section that Cartesian product has the effect of doing subtyping. Therefore, it becomes very apparent that using attributes to derive the generalization, i. e., subtyping, relationship is a very natural approach.

4.5. Class Hierarchy in TOM

The hierarchy of all classes in The Object Modeling is shown in Figure 4.5.1. In this Figure, we see that the universal type is always the toppest level type which is the supertype of all other types. Two major subtypes of the universal type are the record type and the table type. Under the record type, there are many more subtypes to be defined. The same situation applies to the table type.

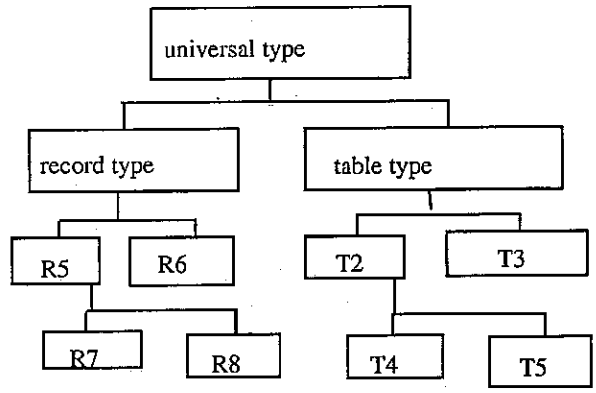

Figure 4.5.1. Hierarchy of TOM Classes

5. INTEGRATION CHARACTERISTICS OF TOM

The most significant contribution of The Object Modeling is in its providing a framework which is able to integrate (1) the database manipulation language with the host language, (2) parametric polymorphisms with subtyping ones, (3) universal polymorphisms with overloading ones, and (4) generic functions with inheritance. Let us study them in more detail now.

5.1. Integration of the Data Manipulation Language and Host Language

General programming languages, e. g., C, Pascal, C++ [Stro86], and Ada [Ada84] are host languages (HL.) They are used for decisions, for displaying questions, and for reading answers. Operations on the database require a specification, called a data manipulation language (DML) or query language, in which commands are expressed to access data from databases in a very efficient way.

The DML/HL dichotomy is generally considered an advantage rather than a deficiency in database systems [Ullm88]. However, there are some new applications of database systems that do not follow the older paradigms, and in this applications, the integration of the data manipulation language and host languages becomes important.

It is TOM which makes this kind of integration possible. In TOM, two kinds of class are provided. Record class is for defining the user object while table class is for the database object definition.

5.2. Integration of Parametric Polymorphisms with Subtyping ones

In conventional object oriented modeling, parametric polymorphisms are treated differently from subtyping polymorphisms [Card86]. The subtyping polymorphism can be very easily achieved through TOM. The inheritance techniques achieved through adding attributes belong to this kind of polymorphism.

Let us see an example to show how the parametric polymorphism can be achieved using TOM.

```
class_name:     R1
super_class:    Record
attributes:     *  a1
                .....
methods:        type1   m1 (a1) body1
                .....


class_name:     R2
super_class:    R1
attributes:     (R1.a1)↓integer     a2
                .....
methods:        .....
```

In the above definitions of R1 and R2, we know that R2 is the subtype class of R1. Attribute "a1" is defined as a generic type and method "m1" is using "a1" as its input parameter. For the "R2" definition, it has an attribute "a2" which is an integer subtype of "a1." (Note that "a1" is a generic type and "a1" and "a2" are using the same cardinal position.) The class R2 does not need to define method "m1" any more. R2 can use it directly.

As the difference between the parametric and subtyping polymorphisms, we find that for the subtyping one the inherited method is defined on those attributes which are not subtyped by the lower level class. On the other hand, for the parametric polymorphism then the inherited method is defined on those attributes which are subtyped by lower level classes. However, we find this difference to be insignificant. Therefore, we easily integrate these two kinds of polymorphism into one.

5.3. Integration of Universal Polymorphisms with Overloading Ones

In their paper [Card85], Cardelli and his co-author Wegner claim that overloading is not a true, or say it ad hoc, polymorphism. They especially separate it from the universal polymorphisms, e. g., parametric and subtyping.

We find it unnecessary to do so. In the cardinal typing system. an overloaded name can be modeled by the "+" constructor. For example, let see the following class definition.

```
class_name:     R1
super_class:    Record
attributes:     integer+boolean+real     a1
methods:        .....


class_name:     R2
super_class:    R1
attributes:     (R1 a1)↓integer     a2
methods:        .....
```

In the above example, the type of attribute "a1" is overloaded with three different types, i. e., integer, Boolean, and real. This is totally legitimate in the TOM modeling. In "R2" class, attribute "a2" is subtyped from the type of "a1." Therefore, "R2" is a subclass of "R1." In this matter, modeling overloading is absolutely no different from modeling any other polymorphism.

5.4. Integration of Generic Functions with Inheritance

In conventional object-oriented modeling such as C++ the generic function is treated separately from the inheritance [Stro86]. This will make the object-oriented modeling, which is the central role of an object-oriented development, very ackward.

Recall from the discussion in Section 5.2, we find the example shown there is not only a parametric polymorphism but also a generic function. Although we call the above example a parametric polymorphism. However, it apparently constructs the basic concept of generic functions. From this point, we find a way of integrating the parametric polymorphism with the generic function.

6. CONCLUSIONS

In this paper, we present a model which completely follows the framework of good type theory practice. By doing so, this model is able to clarify a lot of mis-understandings which occur in other conventional object oriented modeling. We have shown TOM's capability to integrate (1) the database manipulation language with the host language, (2) parametric polymorphisms with subtyping ones, (3) universal polymorphisms with overloading ones, and (4) generic functions with inheritance.

As the implementation of TOM, we have already submitted a proposal to the National Science Council of Republic of China. Hopefully, it will be granted. If everything goes well, the implementation, temporarily called Metadata Design and Implementation of The Object Modeling [Chao93], will be on the way immediately.

Also, the syntax presented in this paper is not formally specified. It will also be the key target once the implementation of TOM starts.

7. REFERENCES

[Ada84] ANSI/MIL-STD-1815A. Military Standard. *Ada Programming Language.* Ada Joint Program Office, U.S. Department of Defense.

[Booc87] Booch, G. *Software Components with Ada. Structure, Tools, and Subsystems.* Benjamin/Cummings. 1987.

[Card85] Cardelli, L. et al. *On Understanding Types, Data Abstraction, and Polymorphism.* Computing Surveys 17: 4. 1985.

[Chao88] Chao, W. *Denotations Semantics-Directed Compiler Generation for Data Flow Machines.* Ph. D. Dissertation. University of Alabama at Birmingham. 1988.

[Chao91] Chao, W. and C. Naden. *TOL: The Object Language.* Company Internal Report. FacStore, Inc. New Jersey. 1991.

[Chao93] Chao, W. *A True Mechanism of Overloading and Its Type Checking Resolutions.* Submitted for Publication. 1993.

[Chao93] Chao, W. *Metadata Design and Implementation of The Object Modeling.* Paper in Preparation. 1993.

[Cox86] Cox, B. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley. 1986.

[Dahl72] Dahl, O. et al. *Structured Programming.* Academic Press. 19722.

[Fiel88] Field, A. *Functional Programming.* Addison-Wesley. 1988.

[Gold89] Goldberg, A.and D. Robson. *Smalltalk-80. The Language.* Addison-Wesley. 1989.

[Lisk80] Liskov, B. Programming with Abstract Data Types. In *Programming Language Design.* Computer Society Press of IEEE. 1980.

[Meye88] Meyer, B. *Object-Oriented Software Construction.* Prentice-Hall. 1988.

[Stro86] Stroustrup, B. *The C++ Programming Language.* Addison-Wesley. 1986.

[Reyn85] Reynolds, J. Three Approaches to Type Structure. Lecture Notes in Computer Science. Vol. 185. Springer-Verlag. pp. 97-138.

[Rumb91] Rumbaugh, J. et al. *Object-Oriented Modeling and Design.* Prentice-Hall. 1991.

[Ullm88] Ullman, J. *Database and Knowledge-base Systems.* Vol.1, Computer Science Press. 1988.

[Wegn88] Wegner, P. *The Object-Oriented Classification Paradigm.* In Research Directions in Object-Oriented Programming. The MIT Press. 1988.