HANA: A Model of Type and Inheritance for Object-Oriented Programming Languages

Keung Hae Lee and *Dennis Kafura*

TR 90-7

HANA: A Model of Type and Inheritance for Object-Oriented Programming Languages

Keung Hae Lee Dennis Kafura

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

Abstract

Most current object-oriented languages consider inheritance as subtyping. However, a type system which views inheritance as subtyping allows neither multiple representations of a type nor method exclusion without violating typing constraints. These problems and the difference between inheritance and subtyping have been well recognized recently. Another approach to subtyping found in the literature is to separate inheritance from subtyping. In this approach, subtyping is solely determined by interface conformance. While subtyping based on interface conformance can support multiple representations, it cannot distinguish intended conformance from accidental conformance. This paper describes a new model of type and inheritance, called HANA, in which subtyping and inheritance are not separated, although differentiated. The notion of subtyping in HANA is based on both inheritance and interface conformance. HANA integrates multiple inheritance, multiple representations, method exclusion, and method name overloading with static typing. HANA extends other existing models of inheritance and subtyping with enhanced expressive power, increased reusability, and program efficiency. We show that the differentiation made by HANA between inheritance and subtyping allows name collision to be resolved without compromising the integrity of the type system. The capability of excluding an inherited method without violating static typing offers a sound solution for resolving name collision in multiple inheritance when used together with other mechanisms. We show that the mechanisms of method exclusion, addition, and renaming are orthogonal with respect to the power of resolving name collision in multiple inheritance.

1 Introduction

Many object-oriented languages which support inheritance and subtyping consider inheritance and subtyping identical. However, this view makes it difficult to support exclusion of an inherited method and multiple implementations of a type. These problems and the difference between inheritance and subtyping have been well recognized recently [LaLonde 86, Snyder 86A, America 87, Cook 89].

Method exclusion is motivated by the reuse of existing type definitions. However, when a subclass excludes an inherited method from its interface, a type error may occur at run time. Consider the following code:

```
class A { f() {...}, ... };
class B { inherit A; exclude A.f(); ...}
a1:A;
a1 := B.create(...);
a1.f();                    /* dynamic type error */
```

While the invocation of a1.f() is correct from the typing standpoint, the above sequence of statements causes f() to be applied to an instance of B, which does not understand the message. Hence, an error occurs at run-time. Two approaches to dealing with method exclusion is that of CommonObjects [Snyder 86B] and exception mechanisms [Borgida 86, Wegner and Zdonic 88]. CommonObjects supports method exclusion by allowing a class to be a subtype of a class which is not its ancestor in the class hierarchy. Exception mechanisms also allow a class to delete an inherited method. An invocation of a deleted method is viewed as an exception, which triggers a certain recovery action rather than causing an error. While Sina/st [Aksit and Tripathi 88] also supports method exclusion, the language has no notion of subtyping.

Many examples can be found which argue for the flexibility of having several representations of a type, e.g., a regular matrix and a sparse matrix, locations represented in rectangular coordinates and polar coordinates, files stored in disk and tape. A variety of techniques have been used to support multiple representations. In Exemplar Based Smalltalk [LaLonde 86], a class may have different exemplars which are prototype instances having different implementations. However, Exemplar Based Smalltalk provides only run-time type checking like Smalltalk. [Decouchant 89] describes a language which separates types from classes. A type can be implemented by several different classes. Duo-Talk also supports multiple representations by separating types from classes [Lunau

89]. However, Duo-Talk has no class inheritance. [Canning 89] separates the type hierarchy from the class hierarchy where class is not a type. A type can be implemented by multiple classes. There exist two languages which support multiple representations without inheritance. Emerald [Black 87], a statically typed language, provides multiple representations by basing subtyping on only interface conformity. No inheritance is supported by Emerald. A delegation based language Act1 [Lieberman 87] relies on message passing and run-time binding to support multiple representations. Since a variable is not typed, objects having different implementations can be assigned to the same variable.

From these examples, we can make two observations. One observation is that method exclusion is a problem only when subtyping and subclassing are equated. The other observation is that multiple representation is a problem only when a class is both a type and an implementation at the same time. It is impossible to associate a type with multiple implementations if every type is defined by a class.

In this paper, we describe a new model of type and inheritance, called HANA, intended for a statically typed language. It supports multiple inheritance, method exclusion, and multiple representations. In HANA, subtyping is based on both inheritance and interface conformance. Hence, HANA distinguishes subtyping from inheritance, but does not separate the type and class hierarchies. Rather, a new kind of type, called *abstract type,* is introduced for the support of multiple representations.

Our decision to base subtyping on inheritance is motivated by a semantic reason and an implementation concern. While Emerald, whose subtyping is solely based on the signature compatibility, can support subtype polymorphism and multiple representations, signature compatibility is weak in capturing semantic relationships among types as observed by [America 87, Liskov 87]. Consider a stack which supports two methods get() and put(Integer) and a queue whose interface also consists of these two methods. A stack certainly is not a queue even if they are signature compatible. Hence, it is possible that two signature compatible types are actually incompatible types. As [Snyder 86A, America 87] explains, a complete solution would be possible only if formal semantic specification techniques are included in a programming language. [Wegner and Zdonic 87] similarly observe that there are levels of behavior compatibility, some of which go beyond signature compatibility. However, since the current techniques of formal specification cannot capture all aspects of object behaviors, we need to turn to other alternatives. The expressive power of Emerald subtyping is limited since all types having the same interface are effectively

collapsed into a single type. Although it is not a perfect solution, inheritance offers a middle ground since with inheritance, the difference between stacks and queues can readily be expressed.

The second concern is the cost of supporting multiple representation. While multiple representation provides programming flexibility, its disadvantage is the extra cost associated with method binding. While basing type compatibility on class inheritance may reduce flexibility, method binding can be implemented efficiently [Snyder 86A, Stroustrup 87, Lee and Kafura 89]. For example, a language like C++ uses static binding, and even when late-binding is needed, it can be done efficiently. In the presence of multiple representations, this implementation advantage is difficult to exploit. Implementation schemes for a type system supporting multiple representations are described in [Black 87, Connor 89]. In both schemes, method binding involves extra mapping between the type of a variable and the type of the object which might be assigned to that variable.

While HANA supports multiple representations, it pays the cost of expensive binding only where the multiple representation feature is used. At an extreme, if all types used in a program have only single implementations, the type system reduces to the one which supports no multiple representations and so no overhead exists. The additional overhead incurred by the use of multiple representation is relatively small if a program is mostly based on types having single implementations.

HANA supports method exclusion as a type definition mechanism without compromising the integrity of the type system. An immediate advantage offered by method exclusion is the increased reusability through selective inheritance. The capability of excluding an inherited method is also useful as a mechanism for resolving name collision in multiple inheritance. It will be shown that the distinction between inheritance and subtyping as used by HANA loosens the difficulty of the name collision problem. Method exclusion together with method overriding and renaming offer a viable solution for the name collision problem in multiple inheritance.

2 An Inheritance Model

There are two kinds of types in HANA, namely, *abstract type* and *concrete type*. An abstract type is a specification of an *interface*, that is, the set of operations applicable to its objects. Each operation in the set is denoted only by a method name and its signature. We will use the term method specification to refer to its name and signatures. An abstract type

definition is only an interface and has no attached implementation. In contrast, a concrete type, which is a class, is a combination of an interface and a representation. Each class definition yields a new concrete type.

An abstract type may inherit from other abstract types. Similarly, a class may inherit from other classes. The inheritance between abstract types is called *interface inheritance* while the inheritance between concrete types is called *class inheritance*. Hence, two distinct inheritance hierarchies exist in HANA: an *interface hierarchy* and a *class hierarchy*. HANA assumes multiple inheritance in both interface inheritance and class inheritance. In order to pursue the idea of separating interface hierarchy from class hierarchy, the discussion in this section will assume that no name clashes occur in the definition of a type. The issue of name conflicts will be considered in Section 3.

2.1 Interface Inheritance

An abstract type may inherit methods from other abstract types, add new methods, or delete inherited methods. The definition of an abstract type T has the form:

```
Interface T {
        inherit      S1,S2,...,Sk;
        exclude      f1,f2,...,fm;
        add          g1,g2,...,gn;
}
```

The set of methods in T is computed by the following set algebra:

$$T = S1+S2+...+Sk - \{f1,f2,...,fm\} + \{g1,g2,...,gn\}$$

where "+" and "-" denotes set union and set subtraction, respectively.

2.2 Class Inheritance

A class definition may inherit from other classes, delete some inherited methods, adding new methods, or rename inherited methods Furthermore, a new method overrides all inherited methods that have the same name and signature. This method overriding rule is different from that of other languages which define it in terms of conformance [Schaffert 86, Meyer 88]. This new overriding rule is motivated by HANA's support of method name overloading [Lee and Kafura 90]. In HANA, the overriding rule of other languages can be expressed using both exclusion and addition of methods. An example class definition is shown in Figure 1.

```
Class T1 {
        inherit C1,C2;
        exclude C1.f, C2.f2;
        add
                f3() { ...}
}
```

Figure 1 - Definition of Class T1

The set of methods in the interface of C1 is computed by

$$T1 = (C1 - \{f1\}) + (C2 - \{f2\}) + \{f3\}$$

where +, and - denote set operations.

2.3 Typing

HANA is intended for a statically typed language. In such a language, the type of each expression would be determined by the analysis of the program text such that if a program is accepted by the compiler, no object will ever be requested to perform an unknown operation. As with most other statically typed languages, the static type-checking is based on explicit type declarations of each identifier.

In HANA, every object is an instance of some class, which determines the object's type.

> The type of an object is the class of which it is an instance.

The compatibility of assignment is defined by subtyping:

> An object or variable of type X can be assigned to a variable of type Y if and only if X is a subtype of Y.

Argument passing is defined similarly.

2.3 Subtyping

We now define the subtype relation. In HANA, subtyping is based on two conditions: *inheritance* and *conformity*. The subtype relation between two types is defined as follows:

A type X is a *subtype* of another type Y if and only if one of the following holds:
(1) X and Y are identical
(2) X *inherits from* Y and X *conforms to* Y.

Further, if X is a subtype of Y, we call Y a supertype of X. The definition of conformance follows.

Type X *conforms to* type Y if and only if
for any method y in Y, there exists a method x in X such that x conforms to y.

The conformance between two methods is defined by the well-known rule of contravariance [Cardelli 84]:

Method f(S1,S2,...,Sn):T *conforms to* method f(U1,...,Un):V if and only if
type Si is a supertype of Ui for all i in {1,...n} and type T is a subtype of V.

According to the HANA definition of subtype, a type X conforming to another type Y is not always a subtype of Y. Furthermore, a subtype relation is not automatically established by inheritance. Since the subtype relation is a more constrained form of the inheritance relation, the subtype relation is a subset of the inheritance relation and also a subset of the conformance relation. In general, the type hierarchy is embedded within the inheritance hierarchy, which contains types related not only by subtype but also by inheritance with no conformance.

2.4 Abstract Types vs Class Types

Multiple representation means that if two different classes implement the same abstraction, their instances can be assigned to the same variable or passed as arguments of the same method, even if no inheritance relation exists between them. In HANA, multiple representation is supported by abstract types. An abstract type may have several different implementations. Each implementation of an abstract type is a class. In order to indicate which classes implement a given abstract type, we define a mechanism which links an abstract type and a class. A class definition can specify an abstract type in its inheritance list. In this case, we say that the class inherits a specification from the abstract type. Further, this kind of inheritance is called *specification inheritance.* In contrast with interface inheritance and class inheritance, specification inheritance only associates a class with an

abstract type of which the class is an implementation. The interface of the class must conform to the specified abstract type. Since a class inheriting from an abstract type always conform to the abstract type by definition, the class is a subtype of the abstract type. When a class inherits from more than one abstract type, the class is a subtype of each of these abstract types.

A variable or parameter declaration can use either an abstract type or a concrete type (class) as a type designator. There is a significant difference between declaring a variable as a class type and as an abstract type.

For example, consider the inheritance hierarchy and a program shown in Figure 2. X is an abstract type and Y and Z are two concrete types implementing X, while W is a subtype of Y. The variable y may be assigned only the instances of Y or W. In contrast, the set of objects which can be assigned to x includes not only those objects but also instances of class Z.

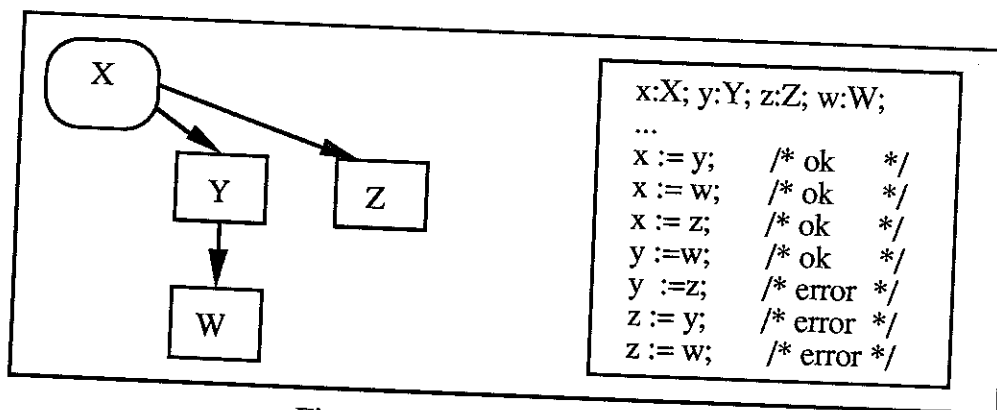

Figure 2 - An Example Program

Two interesting cases exist: an abstract type with no implementations and a class inheriting from no abstract types. An abstract type without an implementation may be useful for classifying behaviors. Such abstract types factorize the common aspects of several similar behaviors.

A class definition with no abstract type in its inheritance list is equivalent to a class found in other object-oriented languages. Some programs will need only a class hierarchy, in which case method invocations can be efficiently implemented [Lee and Kafura 89]. Even when a program needs the support of multiple representations, more expensive binding overhead is

paid only for invocations on abstract type variables. Hence a program can be efficient if class types are dominant in the program.

In HANA, an abstract type can exclude inherited methods, and a class is allowed to exclude or rename inherited methods. The restriction on a class definition is that if the class inherits from an abstract type, the net effect of deletion and renaming must result in a class whose interface conforms to the abstract type.

Figure 3, and 4 show the definitions of an abstract type Stack and its two representations: ArrayStack and ListStack. Class ArrayStack is an implementation using an array. Class ArrayStack has no superclass. We assume that only methods are exported by a class. Hence, ArrayStack exports empty():Boolean, top():Integer, push(Integer), and pop():Integer, which exactly match the set of methods defined by the abstract type Stack.

Class ListStack implements Stack using a list data structure. Let us assume that ListQueue is a class which has an interface of Queue. Since ListStack inherits from ListQueue, it inherits all methods exported by ListQueue. Among these, methods front():Integer and get():Integer are renamed to top():Integer and pop():Integer, respectively. ListStack excludes an inherited method enqueue(Integer) and adds a new method push(Integer):Stack. The resulting interface of ListStack is {empty():Boolean, top():Integer, push(Integer), pop():Integer}, which satisfies the requirement of a Stack interface.

```
Interface Stack {
    empty():Boolean;
    top():Integer;
    push(Integer);
    pop():Integer;
}
```

Figure 3 - Definition of Abstract Type Stack

```
Class ArrayStack is Stack {
        current : Integer;
        store : Array [1..MAX] of Integer;
        empty ():Boolean  { ... }
        top(): Integer  { ... }
        push(item: Integer)  { ... }
        pop(): Integer { ... }
}

Class ListStack is Stack {
        inherit    ListQueue;
        exclude    ListQueue.enqueue;
        rename     ListQueue.front()/top, ListQueue.get()/pop;
        add
                   push(item: Integer) { ... }
}
```

Figure 4 - Definitions of ArrayStack and ListStack

The above example shows the power of differentiating subtyping and inheritance. A class can rename or exclude an inherited method without breaking subtyping constraints. In the above example, while ListStack uses both renaming and exclusion of inherited methods, their use does not cause a typing error since ListStack is not a subtype of ListQueue. Reusability is enhanced because a class can reuse any other classes by selectively inheriting their methods without regard for possible type implications.

3 Multiple Inheritance

There are two reasons why a type inherits from another type. One reason is to create a new type which has a specialized behavior of an existing type. In this case, the new type should conform to the parent. The other reason for inheritance is simply to reuse an existing type definition. If inheritance is motivated only by reuse, a new type may want to exclude some of the inherited methods, thereby rendering itself incompatible with its parent. As described earlier, HANA fully supports inheritance motivated by either of these two reasons.

A difficult issue in supporting multiple inheritance is the name collision that results from a type inheriting a method with the same name from more than one parent. Suppose, for example, that method f() is implemented by two classes X and Y which are the parents of class T. Consider the following scenario:

```
var t: T;
        t = T.create();  /* create an instance of T */
        t.f();           /* which f()? */
```

It is ambiguous which f() should be invoked by t.f(). A similar scenario can be constructed for abstract types. This example is a special case of a more general problem. While we assumed that the two versions of f() have the same signature in the above example, in general, a type may inherit methods which have a common name but different signatures. This paper focuses on the case where conflicting methods have the same signature. Resolution of the general form of name collision requires the support of overloading and is discussed in [Lee and Kafura 90]. Since the semantics of interface inheritance is different from that of class inheritance, the following discussion considers name collision in interface inheritance and that in class inheritance separately.

Name Collision in Abstract Types

Resolving name collision in interface inheritance is simple. An abstract type is merely an interface whose methods have no attached implementations. Since two methods with the same specification are not distinguishable, only one of them is contained in the type. Our treatment of an interface as a set of methods correctly captures this idea. Hence, the following rule, which we will refer to as *automatic merging*, is used for abstract types.

> If an abstract type inherits two methods with an identical specification, the type contains only one of them

Name Collision in Concrete Types

Our model assumes a complete encapsulation of a class. An instance variable defined in a class can be accessed only through its methods. We will assume that for each instance variable, a pair of operations, one for reading and the other for writing the variable, are automatically provided. In such a system, name collision involves only method names, and hence, it is sufficient to consider only collision between method names.

There are at least two serious problems which makes automatic merging undesirable in class inheritance. On the one hand, two methods with the same specification may be two different operations which have the same specification by accident. In this case, merging the two methods will cause one of them to be excluded from the subclass. The implication of excluding an inherited method will be discussed further in this section. On the other

hand, even when they actually denote the same implementation inherited twice via different paths, automatic merging breaks the encapsulation of the superclasses by making the use of inheritance in the superclass visible to the subclass. This problem was first noticed by Snyder [Snyder 86A]. For this reason, CommonObjects explicitly forbids a class to inherit two methods with the same name regardless of their implementations.

Since it is clear that automatic merging cannot be used for class inheritance, it is necessary to resolve name collision explicitly in order to bind each method invocation to a unique method. There are three possibilities:

1. Method Exclusion
2. Method Overriding
3. Method Renaming

Languages can be found which use each of the three mechanisms. Method exclusion is implicitly used by Flavors, CommonLoops, and Sina/st, in which an ambiguous invocation is always bound to a fixed method among the conflicting methods. They use a precedence rule to determine a unique method. Multiple inheritance in C++ uses method overriding, while Eiffel uses method renaming. While a language may use one or more of these three approaches, there are subtle but significant differences in their use. These mechanisms are orthogonal and resolve name conflict in different ways. We will use the following definitions of class X and Y for the comparison of the three approaches:

```
Class X {
        ...
        f() { ... };
}

Class Y {
        ...
        f() { ... };
}
```

Resolution by Method Exclusion

In this approach, a child class may exclude any inherited methods so that every method name in its interface becomes unique. For example, consider the following definition:

```
Class T {
        inherit X, Y;
```

```
        exclude X.f();
        ...
}
```

It is important to know that T should not be a subtype of X because a serious problem arises in this case. Assuming that the above definition makes T a subtype of X, consider the following program.

```
t:T; x:X;
t = T.create();
x = t;
...                /* invocations of X's methods on x */
x.f();             /* invocation of Y.f() */
```

Since X.f() was excluded from T's interface, the f() applied to an instance of T always invokes Y.f(). It is not possible to invoke X.f() on the instances of T. However, other methods of X are still available in the interface of T. If the method f() in X and Y are implemented by different classes, the use of Y.f() together with other methods of X on the same object may break invariants assumed by the methods of X. Hence, this use of subtyping will cause objects named by x to be inconsistent. The method intended by x.f() is X.f(), not Y.f(). This example shows that T should not be considered as a subtype of X. It is not possible to define T to be a subtype of both X and Y with method exclusion. Hence, method exclusion alone does not completely solve the problem of name collision.

The mechanism of method exclusion is useful when a subclass wants to inherit methods of a superclass selectively. In this situation, the subclass is not and does not need to be a subtype of the superclass.

Resolution by Overriding

The second approach is to define a new method in the subclass which overrides conflicting methods. Continuing with the above definitions of X and Y, consider the following definition of T.

```
Class T {
        inherit X, Y;
        f() { Y.f(); };  /* override f() defined in X and Y */
                ...
}
```

In this definition, T defines its own f() which overrides X.f() and Y.f(). The new method f() simply invokes Y.f() so that an invocation of f() on the instances of T always have a unique binding. Since the new f() defined by T overrides f() of both X and Y, T conforms to both X and Y, and should be considered as a subtype of both. While the inconsistency problems discussed above might occur, we view this as what the programmer intends. Otherwise, method exclusion should be used instead of method overriding. While the new f() in the above example simply invokes Y.f(), it may have any definition. For example, the new f() might invoke both X.f() and Y.f(), which is not possible with method exclusion.

Although method overriding offers some flexibility which method exclusion does not, method overriding cannot replace method exclusion. Method overriding alone cannot solve the problem of an inconsistent object, which is resolved by method exclusion. However, these two mechanisms are not sufficient to completely solve the name collision problem. Neither of these allows the desired version of f() to be selected.

Resolution by Method Renaming

Method renaming solves the last problem. With method renaming, T might be defined as follows:

```
Class T {
        inherit X, Y;
        rename X.f()/myf;       /* rename X.f() by myf */
        ...
}
```

We will show that both X.f() and Y.f() can be invoked on the instances of T, and that the inconsistency problem does not exist with this approach. Consider the following code which uses this definition of T. The number at the left of each line is a line number.

```
1       x:X; y:Y; t:T;
2       t = T.create();
3       x = t;            /* T is a subtype of X */
4       y = t;            /* T is a subtype of Y */
5       t.f();            /* invoke Y.f() */
6       t.myf();          /* invoke X.f() */
7       x.f();            /* invoke X.f() */
8       y.f();            /* invoke Y.f() */
```

The call t.f() in line 5 invokes Y.f(). Note, however, that X.f() can also be applied to an instance of type T as shown in line 6. Since X.f() has been renamed to myf within T, one can refer to X.f() through the new name myf.

The beauty of the renaming mechanism is demonstrated in line 7. Note that x.f() is bound to X.f() rather than Y.f(). If an instance of T is treated as an object of type X so that all operations applied to that object may be those of X, the other two mechanisms do not help. Obviously, method renaming can not replace exclusion and overriding.

In summary, we have shown that the mechanisms of method exclusion, overriding, and renaming are orthogonal to each other in resolving name collision in multiple inheritance. These mechanisms can be efficiently implemented if late binding of methods is based on indexing into a run-time table.

4 Conclusion

While the difference between inheritance and subtyping has been well recognized recently, it is less obvious how they should be distinguished. In this paper, we described a new model of type and inheritance, called HANA. HANA is different from other proposals which also recognize the difference between inheritance and subtyping in several aspects. HANA differentiates subtyping and inheritance rather than separating them. The notion of the abstract type allows HANA to be implemented efficiently, while providing the flexibility of multiple representation. The cost of expensive binding incurred by multiple representation is paid only when it is actually used. It is possible to distinguish intended conformance from accidental conformance since class hierarchies are constructed explicitly. Another advantage is the support of method exclusion as a type definition mechanism without violating typing constraints. This capability of method exclusion, when used together with method overriding and renaming, offers a viable solution to the name collision problem in multiple inheritance.

5 References

[Aksit and Tripathi 88] Mechmet Aksit and Anand Tripathi, Data Abstraction Mechanisms in Sina/st, OOPSLA '88 Proceedings, 1988.

[America 87] Pierre America, Inheritance and Subtyping in a Parallel Object-Oriented Language, ECOOP '87 Proceedings, LNCS, Vol 276, Springer-Verlag, 1987.

[Black 87] Andrew Black, Norman Hutchinson, et al., Distribution and Abstract Types in Emerald, IEEE Trans. Soft. Eng., Vol SE-13, No. 1, 1987.

[Bobrow 86] Daniel G. Bobrow, Kenneth Kahn et al., CommonLoops:Merging Lisp and Object-Oriented Programming, OOPSLA '86 Proceedings, 1986.

[Borgida 86] Alexander Borgida, Exceptions in Object-oriented Languages, SIGPLAN Notices, Vol 21, No 10, October 1986.

[Canning 89] Peter Canning, William R. Cook, et al., Interface for Strongly-Typed Object-Oriented Programming, OOPSLA '89 Proceedings, 1989.

[Cardelli 84] Luca Cardelli, A Semantics of Multiple Inheritance, LNCS Vol. 173, Springer-Verlag, 1984.

[Connor 89] R.C.H. Connor, A. Dearle, et al., An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance, OOPSLA '89 Proceedings, 1989.

[Cook 89] William R. Cook, Walter L. Hill, et al., Inheritance Is Not Subtyping, Conference Record of Seventh Annual ACM Symposium on Principles of Programming Languages, 1989.

[Decouchant 89] D. Decouchant, S. Krakowiak, et al., A Synchronization Mechanism for Typed Objects in a Distributed System, SIGPLAN Notices, Vol 24, No 4, April 1989.

[Kafura and Lee 89] Inheritance in Actor Based Concurrent Object-Oritented Languages, ECOOP '89 Proceedings, Cambridge University Press, 1989.

[LaLonde 86] Wilf R. LaLonde, Dave A. Thomas, John R. Pugh, An Exemplar Based Smalltalk, OOPSLA '86 Proceedings, 1986.

[Lieberman 87] Henry Lieberman, Concurrent Object-Oriented Programming in Act1. In Object-Oriented Concurrent Programming (edited Akinori Yonezawa and Mario Tokoro), MIT Press, 1987.

[Lee and Kafura 89] Keung Hae Lee and Dennis Kafura, A Fast and Efficient Method Dispatching for Statically Typed Multiple Inheritance Object-Oriented Languages, TR 89-40, Department of Computer Science, Virginia Tech, 1989.

[Lee and Kafura 90] Keung Hae Lee and Dennis Kafura, Overloading Is not Ad Hoc, In Preparation.

[Liskov 87] B. Liskov, Data Abstraction and Hierarchy, OOPSLA '87 Addendum to the Proceedings, 1987.

[Lunau 89] Charlotte Pii Lunau, Separation of Hierarchies in Duo-Talk, Journal of Object-Oriented Programming, July 1989.

[Meyer 88] Bertrand Meyer, Object-Oriented Software Construction, Prentice-Hall, 1988.

[Moon 86] David A. Moon, Object-Oriented Programming with Flavors, OOPSLA '86 Proceedings, 1986

[Schaffert 86] Craig Schaffert, Topher Cooper, et al., An Introduction to Trellis/Owl, OOPSLA '86 Proceedings, 1986

[Snyder 86A] Alan Snyder, Encapsulation and Inheritance in Object-Oriented Programming Languages, OOPSLA '86 Proceedings, 1986

[Snyder 86B] Alan Snyder, CommonObjects: An Overview, SIGPLAN Notices, Vol 21, No 10, October 1986.

[Stroustrup 87] Bjarne Stroustrup, What Is Object-Oriented Programming?, ECOOP '87 Proceedings, LNCS Vol 276, Springer-Verlag, 1987.

[Wegner and Zdonic 88] Peter Wegner and Stanley Zdonik, Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like, ECOOP '88 Proceedings, LNCS Vol 322, Springer-Verlag, 1988.