Inheritance as an Incremental Modification Mechanism
or
What Like Is and Isn't Like

Peter Wegner and Stanley B. Zdonik
Department of Computer Science
Brown University, Providence, RI 02912

Abstract: Incremental modification is a fundamental mechanism not only in software systems, but also in physical and mathematical systems. Inheritance owes its importance in large measure to its flexibility as a discrete incremental modification mechanism. Four increasingly permissive properties of incremental modification realizable by inheritance are examined: behavior compatibility, signature compatibility, name compatibility, and cancellation. Inheritance for entities with finite sets of attributes is defined and characterized as incremental modification with deferred binding of self-reference. Types defined as predicates for type checking are contrasted with classes defined as templates for object generation. Mathematical, operational, and conceptual models of inheritance are then examined in detail, leading to a discussion of algebraic models of behavioral compatibility, horizontal and vertical signature modification, algorithmically defined name modification, additive and subtractive exceptions, abstract inheritance networks, and parametric polymorphism. Liketypes are defined as a symmetrical general form of incremental modification that provide a framework for modeling similarity. The combination of safe behaviorally compatible changes and less safe radical incremental changes in a single programming language is considered.

1. Introduction

Incremental modification facilitates reusing a conceptual or physical entity in constructing an incrementally similar one. It arises in incremental problem solving, incremental specification, incremental compilation, and incremental editing. Evolution of both natural and computational systems may be described and controlled by incremental modification mechanisms. Recursive specifications are dynamically incremental in that they specify the solution of a problem with parameter (n+1) in terms of the same problem with parameter n.

Inheritance is a particular kind of incremental modification mechanism that transforms a parent entity P with a modifier M into a result entity R = P+M. as in Figure 1. The parent, modifier, and result have a record structure with a finite number of attributes:

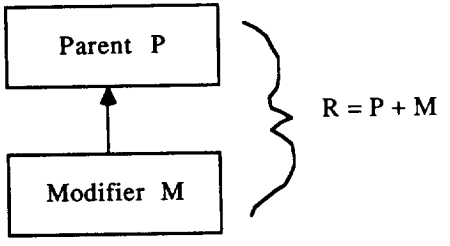


Figure 1: Incremental Modification by Inheritance

$P = (P1,P2,\ldots,Pp)$
$M = (M1,M2,\ldots,Mm)$
$R = (R1,R2,\ldots,Rr)$

The composition operator + is asymmetrical since the parent P and modifier M play different roles in the composition process. The asymmetric role of P and M in determining R is brought out by the following notation:

result R
  inherits P;
  modified by M;

The attributes of M may be independent of or overlap with those of P:

(1) **Independent Attributes:** The attributes of M are disjoint from those of P.
R has p+m attributes consisting of the union of those in P and M.

(2) **Overlapping Attributes:** The attribute names of M overlap those of P.
R has the attributes explicitly added in M and those attributes of P whose names do not occur in M. The attributes of M take precedence over those of P much as identifiers declared in an inner textually nested module of a block-structure language take precedence over those declared in an outer module.

Incremental modification is clearly simpler for independent than for overlapping attributes. However, software evolution generally requires modifying existing attributes of an entity rather than merely adding new ones. We therefore investigate constraints on the incremental modification of attributes, including very strong constraints such as behavioral compatibility and very weak constraints such as cancellation of attributes.

This work is part of a wider study [Wei] which examines the following attributes of inheritance:

modifiability: How should modification of inherited attributes be constrained?
granularity: Should the unit of inheritance be classes or instances, entities or attributes?
multiplicity: How should multiple inheritance be managed?
quality: What should be inherited? Specifications, code, or both?

We focus narrowly on incremental modification, recognizing that an inheritance mechanism requires design decisions for modifiability to be supplemented by decisions concerning granularity, multiplicity, and quality.

2. The Essence of Inheritance

Since our models of incremental modification depend on general assumptions about the nature of inheritance and on the kinds of entities that inherit, we characterize these notions more precisely. Inheritance is discussed first, to bring out the nature of inheritance hierarchies before committing to specific kinds of inherited entities.

What is the essence of object-oriented inheritance? It is a hierarchical incremental modification mechanism for entities defined by sets of attributes that allows any R = P + M defined as in section 1 to be further modified, say to R1 = R + M1. We assume further that inherited attributes are more essentially part of the inheriting object than attributes that are merely used or invoked, just as eye colors are more essentially part of people than the car

which they drive or the house in which they live.

We adopt the view of Cook [Co1, Co2] who defines inheritance as a composition mechanism that internalizes inherited attributes by late (execution-time) binding of self-reference to the inheriting object. Self-reference in a type or class is bound to the object on whose behalf an operation is being executed, rather than to the textual module in which the self-reference occurs.

Dynamic binding of self-reference at execution time captures the essential difference between inheritance and invocation. Late binding of "self" allows an inherited entity P to assume the identity of each of the objects that uses its attributes while preserving its textual independence. Its attributes are shared by inheriting entities but behave like indigenous attributes of each inheritor. One advantage of sharing over copying of inherited attributes is that behavior modification of objects inheriting P can be realized by modular modification of the shared parent P rather than by internal surgery on entities that contain copies of P.

The execution-time binding of self-references in a parent P and modifier M to an object X with template R = P+M is illustrated in Figure 2. When an object X receives a message to invoke one of its attributes (operations) it binds self-references in both P and M to the object X.

In the example below, P defines the attributes P1,P2,...,Pp and P1 contains a textual self-reference "self A" to an attribute A. A may be a locally defined attribute Pi of P or a nonlocal attribute Mj defined in the modifier M.

```
P =
  P1: ... self A; ...
  P2: ...
  ... ...
  Pp: ...
```

When "self A" is executed, it invokes a definition of A in P or M according to the following rules:

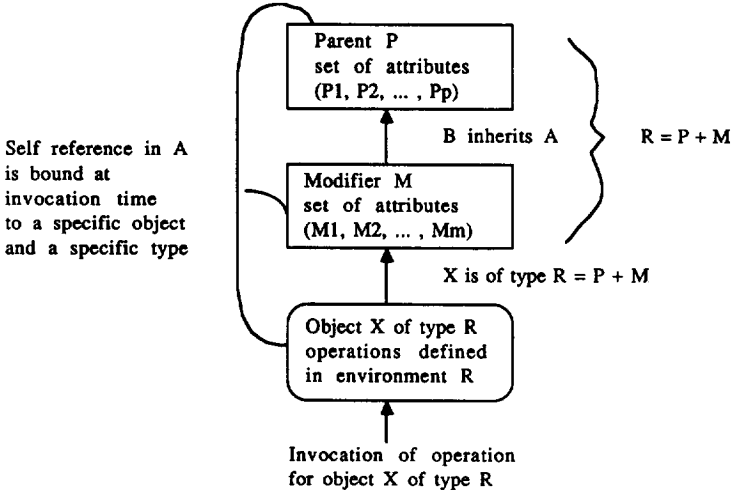


Figure 2: Binding of Self-References in Inheritance

(1) **Redefined attribute:** A is defined in both P and M (A = Pi = Mj for some i,j).
A is bound to the attribute definition Mj in M, which blocks the definition of the similarly named attribute Pi in P. The meaning of A is not the locally defined attribute of P but the nonlocal attribute of M.

(2) **Virtual attribute:** A is defined in M but not in P.
A is called a virtual attribute of P, since it is used but not defined in P. P is called a *virtual entity* (*virtual class* in Simula, *abstract type* in Smalltalk). The definition of P is incomplete in that P must rely on attributes defined in a descendant before objects that use P as a template can be instantiated. Note that if A is defined in M it is bound to the attribute Mj of M independently of whether A is also defined in P.

(3) **Recursive attribute:** A is defined in P but not in M.
A is bound to a locally defined attribute Pi in P. If the attribute A is the same Pi in which the self-reference occurs, then this is a recursive invocation. However, if A refers to another attribute of P then this is a recursive invocation of the object P but not a direct recursive invocation of the attribute definition for A.

Delayed binding of self-reference allows "self A" in P to assume the identity of a variety of different objects at execution time. It allows attributes of a parent to be internalized by the objects that use them in the sense that self-references in the parent refer to the identity of the currently invoking object. It allows parents to specify virtual resources that are not yet defined and to require the definition of such resources in descendant templates.

Attributes of an entity may be locally defined, inherited, or virtual. Inherited and locally defined attributes together determine the resources provided by an entity to its clients. Virtual attributes arise when the provided resources of an entity are insufficient ot fulfil its contract with clients. If "needed resources" > "provided resources" then attributes which are needed but not provided are virtual and must be supplied by a descendant.

The meaning of self-reference in objects, just as in recursive functions and procedures, can be defined in terms of least fixed points. Cook refers to entities with unbound self-references as generators and models binding of self-references in generators P by their least fixed point Y(P). Inheritance is realized by composing the generators P and M to obtain P+M and then taking the least fixed point Y(P+M) of this composite generator. In contrast, invocation is defined in terms of the composition Y(P) + Y(M) of fixed points. Thus Y(P + M) ≠ Y(P) + Y(M), and the difference between the left and right hand sides in fact captures the difference between M inheriting P and M invoking P.

In a world without self-reference, inheritance reduces to invocation and inheritance hierarchies are simply tree-structured resource-sharing hierarchies. However, recursive definitions are just as fundamental for objects as for functions and procedures. The progression from non-recursive Fortran to recursive procedure-oriented languages is being repeated for object-oriented languages. Incremental modification mechanisms should be designed from the start to properly handle self-reference.

3. Types and Classes

The terms "type" and "class" are defined to accord with their current usage. Type and class hierarchies based on these definitions are found to have very different properties.

Types are motivated by type checking and may be defined by a predicate for recognizing expressions of the type. Classes determine collections of objects and may be defined by templates for object creation. Types have a type-checking semantics while classes have an instance creation semantics, as indicated in Figure 3.

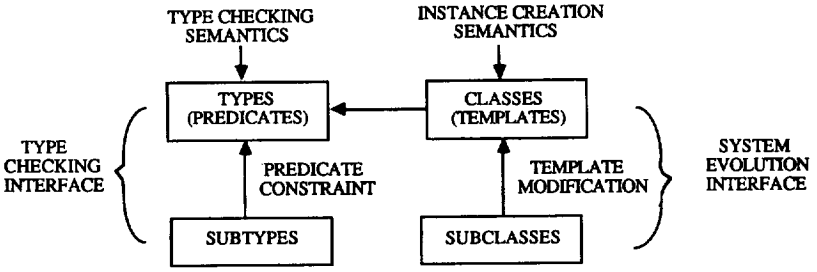


Figure 3: Relation Between Types and Classes

The definition of types as predicates and classes as templates leads to a definition of subtypes in terms of predicate modification and subclasses in terms of template modification. Subtypes are defined in terms of constraints that determine a subset of the set defined by the parent predicate. Subclasses are defined in terms of template modifications that may involve radical modification or even cancellation of template components.

Template modification is more powerful than subtyping as an incremental modification mechanism but also less tractable. Inheritance will be defined as a mechanism for template modification rather than subtyping. This reflects not only the reality of inheritance mechanisms in actual programming languages but also the fact that evolutionary processes of incremental change in the real world rarely conform to the stringent constraints of subtyping and are better modeled by subclassing.

Classes are a special kind of type, namely a type whose predicate is a template specification. For every class C 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 C". However, types do not necessarily have an associated class since predicates do not necessarily specify templates.

Class-based languages automatically have an associated type system. Class declarations may be viewed as type declarations. Class hierarchies of object-oriented languages automatically have associated type hierarchies. For every statement about classes there is a corresponding statement about associated types, but the converse is not necessarily true. The condition "every object belongs to a class" implies that every object also belongs to a type. However, the converse condition, "every object belongs to a type," does not imply that every object belongs to a class.

Both classes and types serve to classify values into collections with uniform attributes. But classes are collections of created or potentially creatable instances, while types focus primarily on the predicate. Every class has two associated collections: the collection of all possible instances of the class and the collection of instances that has actually been created.

Types may be specified syntactically by signature specifications, semantically by behavior specifications, and pragmatically by implementations, as illustrated in Figure 4. Signature specifications in terms of record types are underspecifications since the semantics of typed record components is left unspecified. Behavior specifications in principle precisely determine the desired goal or task realized by an object but cannot be uniformly specified in any formal specification language. Implementations overspecify the behavior by committing to a particular implementation of the behavior. Classes are pragmatic template specifications.

We are interested in incremental modification mechanisms associated with each of the specification techniques:

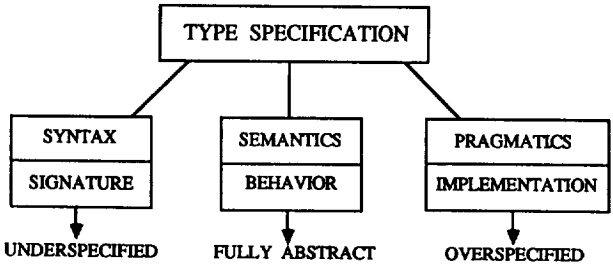


Figure 4: Syntactic, Semantic, and Pragmatic Type Specification

(1) Incremental modifications of behavior are generally behaviorally compatible with the behavior of the parent and are called subtype specifications. The subtype is related to the parent type by an is-a relation. Behavior of types and subtypes can be defined by algebras, but the relation between the algebra of a type and its subtype can be quite subtle.

(2) Incremental modifications of signatures determine compile-time checkable subclasses that are not in general behaviorally compatible with the parent type.

(3) Implementation specifications are more flexible in their support of incremental modification mechanisms than behavior or signature specifications, since template modifications of an arbitrary nature may be specified. Behavior and signature compatible modification may be supported at the level of implementation by constraints on the implementation mechanism. However, incremental modification that violates behavior or signature compatibility may also be supported.

The above analysis of types in terms of their mechanisms for specification yields mechanisms for incremental modification closely related to those that arise in actual object-oriented systems.

4. Varieties of Incremental Modification

The constraints on incremental modification imposed by our assumptions about inheritance, types, and classes leave considerable room for variation.

Four increasingly permissive properties of incremental modification are contrasted in Figure 5: behavior compatibility, signature compatibility, name compatibility, and cancellation. Each has a different conceptual model of entities in the inheritance hierarchy. Behavioral compatibility views entities as behaviors modeled by algebras, signature compatibility views entities as signatures modeled by partial ordering lattices, and name compatibility views entities as implementations modeled by an operational semantics. Cancellation views entities as sets that can be enlarged or restricted by respectively weakening or strengthening the constraints of set membership.

(1) Behavior-compatible modification (types)
The entities to be modified are types whose behavior may be modeled by many-sorted algebras. Syntax is specified by signatures and semantics by interpretations or by equational axioms. Modified behavior of subtypes is specified by behaviorally compatible subalgebras. We examine the notion of behaviorally compatibility and demonstrate that true behavioral compatibility which satisfies the principle of substitution is more restrictive than generally supposed.

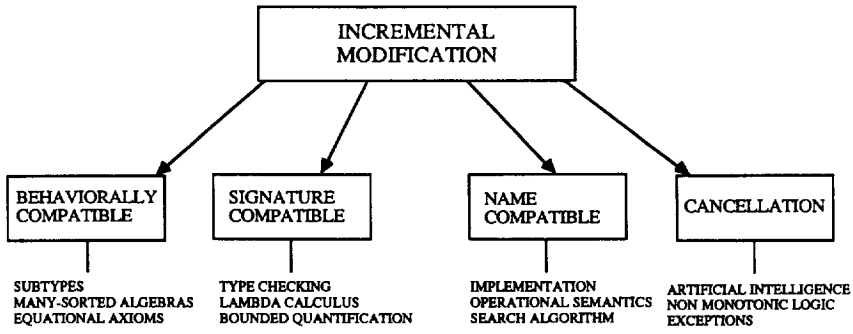


Figure 5: Varieties of Incremental Modification

(2) Signature-compatible modification (signatures)
The entities to be modified are signatures which are syntactic algebra specifications without any associated semantics. They determine a coarser equivalence relation over expressions than behavioral specifications. Subsignatures may be derived by horizontal extension (adding new components) or vertical modification (constraining existing components). Subsignatures are not behaviorally compatible with parent signatures because a given component may be replaced by a type-consistent component with different behavior.

(3) Name-compatible modification (classes)
The entities to be modified are class templates specified by their implementation. Behavior modification is realized by modifier templates which overlay their parent templates. The semantics of modification may be specified operationally by a search algorithm. To find the definition of a named attribute we first look for it in the modifier. If it is not defined in the modifier, we look for it in the ancestor hierarchy, either finding the attribute or reporting that it is undefined.

(4) Inheritance with cancellation (exceptions)
Traditional inheritance usually focuses on subtypes or subclasses defined by increasing the severity of constraints. Cancellation focuses on the relaxation of constraints, thereby allowing extension as well as restriction of sets defined by inheritance. This is reflected in our discussion of exceptions, abstract inheritance networks, and nonmonotonic reasoning systems. Cancellation may occur at the level of behavior, signatures, or names. At the level of names it may be realized by modifying the search algorithm so that search beyond the local level is blocked for cancelled names.

4.1. Behaviorally Compatible Modification

There was much debate in the 1960s whether Algol 68 and Pascal should be behaviorally compatible modifications of Algol 60 and whether IBM System/360 should be a behaviorally compatible modification of the 700 series of computers. Later it was debated whether Ada should be a behaviorally compatible modification of Pascal or simply Pascal-like. In all these cases it was decided that the constraint of behavioral compatibility was too onerous, in spite of the considerable advantages to users of such "upward compatibility".

Behavioral compatibility places strong constraints on incremental modification. When these constraints prevent us from realizing desired incremental changes, then behavioral compatibility must be discarded. But when desired incremental changes are behaviorally compatible, then great benefits can be realized by reusability of both code and concepts.

4.1.1. Specification of Behavior by Algebras

In order to define behaviorally compatible modification, we need a precise notion of behavior. Behavior may be specified by algebras with a signature and a semantics. For single-sorted algebras the signature consists of a sort S and operation symbols fi of arity ni:

Sig = (S; f1:n1, f2:n2, ... , fK:nK)

A signature determines a collection of well-formed expressions for the algebra but does not associate any semantics with expressions. A semantics of an algebra with signature Sig is given by an interpretation that associates a domain (carrier set) of values with S and a function over the domain of arity ni with every function symbol fi. Alternatively, semantics may be associated with a signature by equational axioms such as commutativity and associativity that specify that expressions transformable into each other by axioms are semantically equivalent.

A many-sorted algebra S has a set of sorts S = (S1,S2,...,SN). Function symbols have arguments and values of specified sorts. The interpretation associates a value set with each sort and associates functions from domains to ranges with each function symbol.

An order-sorted algebra is a many-sorted algebra with a partial ordering on its sorts that induces a subtype ordering relation on algebras. A partial algebra is an algebra in which functions may be partial (undefined for some arguments).

Algebras are a precise specification of behavior for classes of objects, with S representing the state and fi representing the attributes of the record structure. They provide a framework for the precise representation of substructure.

4.1.2. Three Kinds of Behavioral Compatibility

Different linguistic notions of subtype yield different notions of behavioral compatibility. We define three successively stronger notions of behavioral compatibility associated with the following three notions of subtype [BW]:

Subset subtype: Int(1..10) is a subset subtype of Int
Isomorphically embedded subtype: Int is an isomorphically embedded subtype of Real
Object-oriented subtype: Student is an object-oriented subtype of Person

4.1.2.1. Subset Subtypes and Partial Compatibility

Subset subtypes need not be closed under their operations. The result of an operation on arguments in a domain may lie outside the domain. We define a form of behavioral compatibility appropriate to subset subtypes called "partial compatibility":

Partial Compatibility: A subtype is partially compatible with its parent type if corresponding operations on corresponding arguments give corresponding results whenever the result is defined for the subtype.

Partial compatibility is not true behavioral compatibility because the result may be defined for the supertype but not for the subtype. For example, "6+7" is undefined for the subtype Int(1..10) but is certainly defined for the type Int.

4.1.2.2. Isomorphically Embedded Subtypes and Subcomplete Compatibility

Isomorphically embedded subtypes are better behaved than subset subtypes, because they are closed under their operations. They may be associated with a different notion of

behavioral compatibility called subcomplete compatibility.:

Subcomplete compatibility: A subtype is subcompletely compatible with its parent type if corresponding operations on corresponding arguments of the subtype are defined in the subtype whenever they are defined in the supertype and yield corresponding results.

Subcomplete compatibility yields true compatibility when operations and arguments are restricted to the subtype, but may result in incompatibility when arguments are extended to include those of the supertype. In particular, the assignment "x := y;" where x is a variable of the supertype and y is an object of the subtype, causes y to be unprotected against inadmissible assignments of supertype values to its components.

If the operations on Int are extended to include division, then Int and Real are no longer subcompletely compatible, since Int is closed only for addition and multiplication and not for division. In mathematics, closure under division provided the motivation for extending the integers to the rationals. Closure under operations has also provided the motivation for further extensions to algebraic, real, and complex numbers. Subcomplete extension is of limited applicability since it cannot admit such partial operations in the subalgebra. Nevertheless, it is useful in practice to view subtypes as being algebras only under operations for which they are closed and to view partial operations of the subalgebra, such as division for integers, as being defined only for the algebra of the supertype.

4.1.2.3. Object-Oriented Subtypes and Complete Compatibility

In order to avoid altogether situations in which operations can be undefined on values of the subtype and defined on values of the supertype, we consider subtypes whose argument domain is precisely the same as that of the parent type. Operations defined for both the subtype and supertype have corresponding values for corresponding arguments over their identical domains. However, the subtype may have additional operations, as is the case for students who are persons with certain specialized operations such as "grade-point average". The notion of behavioral compatibility for such subtypes will be called "complete compatibility".

Complete compatibility: A subtype is completely compatible with its supertype if it has the same domain as the supertype and, for all operations of the supertype, corresponding arguments yield corresponding results.

Complete compatibility yields compatible behavior not only in the context of the subtype but also in the context of the supertype. In particular, it permits values of the subtype to be freely manipulated as values of variables of the supertype with no fear of inadmissible behavior.

The three kinds of behavioral compatibility share the property of yielding the same result when operations on corresponding arguments are defined for both the subtype and the supertype. They differ in the degree to which operations of the subtype may be undefined. Partial compatibility allows lesser definability for arguments in the domain of the subtype, subcomplete compatibility allows lesser definability only in the domain of the supertype, and complete compatibility requires equal definition for the common domain of the subtype and supertype.

4.2. Signature-Compatible Modification

When behavior is too difficult to specify precisely and completely we approximate it by a signature. In particular, type-checking algorithms usually define types in terms of signatures rather than behavior, and define the notion of subtype in terms of compile-time-checkable

constraints on signatures rather than in terms of restrictions on or extensions of behavior.

A semantics-preserving signature-compatible modification is a signature-compatible modification where the attributes of the signature are guaranteed to preserve their semantics provided they remain defined. We define the notions of horizontal extension and vertical modification of signatures syntactically and then show that semantics-preserving horizontal extensions are behaviorally compatible while vertical extensions are not.

4.2.1. Horizontal Extension

A type Person1 with a name attribute of type "String" may be defined as follows:

type Person1 = (name: String);

Person is a horizontal extension Person1 with an attribute "age":

type Person = (name: String, age: Int(0..120))

Horizontal extensions are subtypes that specialize the parent type and have a richer set of attributes than the parent type. The set of objects possessing the richer set of attributes is a subset of the set of objects of the parent type. An increase in the richness of attributes goes hand in hand with a decrease in the number of objects possessing the attributes.

4.2.2. Vertical Modification

The type Person can be vertically extended to the type Retiree as follows:

type Retiree = (name: String, age: Int(65..120);

This vertical extension specializes the name component of Person to a subset of its domain. Checking that a type defined by inheritance is a vertical extension of the type from which it inherits is more difficult than checking for horizontal extension and may in general require dynamic rather than static checking.

Vertical and horizontal extension can be combined:

Let RT1 = (s1: T1)
and RT2 = (s1: T11, s2: T2)

then RT2 is a subtype of RT1
if T11 is a subtype of T1

Vertical modification is particularly common in database systems, although the concept is certainly of more general applicability. The semantic data model (SDM) [HM81] contains a rich subtype definition mechanism that is largely based on this type of relationship. The predicate-defined subclass in the SDM is an example of exactly this type of definition.

Notice that this type of definition allows the type of an object to change as the values of the attributes on which the predicate depends change. For example a person whose age increases beyond 65 suddenly becomes a Retiree.

4.2.3. The Principle of Substitutability

The definition of complete compatibility is motivated by the following principle:

Principle of substitutability: An instance of a subtype can always be used in any context in which an instance of a supertype was expected.

Of our three previously defined notions of behavioral compatibility, only complete compatibility guarantees substitutability. Complete compatibility is realized by semantics-preserving horizontal extension. The following example illustrates that, however, semantics-preserving vertical modification does not guarantee substitutability:

```
p: Person;
r: Retiree;

p := r;
set-age (p, 40);
```

The operator "set-age" works as long as the Retiree type is not defined as a subtype of Person. As soon as this relationship is established, the set-age operation given above fails. The assignment of a Retiree to a variable of type Person (which must be allowed if we wish to maintain the principle of substitutability) has caused a problem: we cannot set the retiree's age to 40. Adding the subtype has contradicted an implicit assertion made in the definition of the type Person. The type Person originally said that the age of a Person can be set to any integer value between 0 and 100. By adding the type Retiree as a subtype, we have said that there are now some persons for whom setting their age to any value between 0 and 64 is illegal.

No restriction of the domain can ever satisfy the principle of substitutability since assignment of a subtype value that is not a supertype value to a supertype variable violates the principle.

4.2.4. Subsets Are Not Subtypes

Should the term "subtype" be used only for subtypes that satisfy the principle of substitutability, or should it be used more broadly to include subset subtypes and isomorphically embedded subtypes? If we restrict the term to its narrow meaning, then subsets are not subtypes. That is, subsets such as Retiree defined by domain constraints are not subtypes because they do not satisfy the principle of substitutability. We say that the relationship between Person and Retiree is not subtype but a subset relationship.

Although the subset relationship does not conform to the principle of substitutability. it can be used for code reusability. If B is-a-subset-of A and A defines a piece of behavior (e.g., a method) that is not defined in B, this behavior can be inherited by B.

The requirement of substitutability and the associated notion of subtype and behavioral compatibility is too strong in many practical situations. It is the subset relation rather than the subtype relation that corresponds to classical is-a relation. The fact that a Retiree is always a Person is in most contexts more important than the fact that certain operations on persons (those appropriate for young persons) are inappropriate when performed on retirees. The restriction that Retirees be behaviorally indistinguishable from young persons is clearly unreasonable. The requirement of substitutability implies such indistinguishability and is therefore unreasonable too. However, for subtypes where such indistinguishability is reasonable, substitutability allows useful compile-time optimizations as well as the ability to evolve code smoothly by adding subtypes that do not break existing programs.

In practice the following weaker substitution principle is often adequate as a basis for behavioral compatibility:

Principle of read-only substitutability: An instance of a subtype can always be used in read-only mode in any context a supertype is expected.

Retirees are read-only substitutable for persons since attributes of retirees are always valid person attributes. Subtypes that are read-only substitutable may be freely used to create new instances, for comparison and discrimination, and in any other supertype context that does not involve assignment.

Note that subset subtypes as well as isomorphically embedded subtypes are read-only substitutable in the context of supertypes. Partial and subcomplete compatibility are both sufficient to ensure read-only substitutability of a subset in the context of the supertype.

4.3. Name-Compatible Modification

Name-compatible modification requires only the name and not the signature of the parent type to be preserved in the result. For strongly-typed languages name compatibility is a more permissive incremental modification mechanism than signature compatibility which allows the type as well as the behavior of arguments to be replaced. However, for non-strongly-typed languages, expressions need not have signatures and name compatibility is the only applicable mechanism.

Whereas signature-compatible modification is specified in terms of predicates for type recognition, name-compatible modification is generally specified in terms of search algorithms for names in a model of implementation:

```
procedure search (name, module)
  if (name = localname) then do localaction
  else if (inherited module = nil) then undefinedname
  else search(name, inheritedmodule)
```

This search procedure describes the basic operational mechanism of searching for a local name and following the inheritance chain if no local name exists. It may be used to implement signature compatibility by introducing a compile-time check which filters out signature incompatible modifications, ensuring that only signature-compatible hierarchies are acted on by the search algorithm at execution time. Behaviorally compatible change may in principle also be implemented by filtering out inadmissible modifications, but checks for behavioral compatibility cannot in general be effectively performed, either at compile time or execution time.

Name compatibility is the simplest form of incremental modification in practical programming languages because it requires no extra checking at compile-time or execution time.

The relation between behavioral, signature, and name compatibility may be summarized by the following characterization of their underlying models:

```
behavioral compatibility: algebraic and axiomatic models
signature compatibility: inclusion relations on domains
name compatibility: model of implementation, search algorithm
```

If we have to choose a single model of incremental modification then name compatibility may well be the best candidate because because it can be flexibly constrained to realize more restrictive forms of compatibility. The models that underlie behavioral and name compatibility cannot be as easily adapted for other purposes.

4.4. Incremental Cancellation

Inheritance with cancellation allows the modifier M to delete as well as modify attributes of the parent P. Attribute deletion is the inverse of horizontal extension and causes the result class to be a generalization of the parent class. It is a symmetrical form of incremental modification that does not distinguish between the creation of subtypes and supertypes. It is the most radical incremental change mechanism specified by inheritance. Examples of attribute deletion include the following:

(1) attribute deletion: The types Ageless-Person or Nameless-Person can be defined by respectively cancelling the attributes age or name from the type Person. Deletion of an attribute may occur when a new attribute causes an already existing attribute to become redundant. For example, adding social security to the type Person causes the name attribute to become redundant so it can be deleted.

(2) exceptions: Sets with exceptions, such as birds which cannot fly, can be handled by the mechanism of cancellation. This is done by defining a basic set of birds which can fly that excludes penguins and ostriches and adding an exception class of non-flying birds by cancellation of the fly attribute. Thus exceptions to be added to a uniformly-defined set may be specified by cancellation of attributes.

Since attributes of a class may refer to or invoke each other, deletion of an attribute may cause a problem if another attribute refers to it. The deleted attribute becomes a virtual attribute that must be defined in a subclass to provide a self-contained object interface.

The search algorithm for name compatibility can easily be adapted to cancellation by adding a test for cancelled attributes.

if (name = cancelledname) then undefined

4.4.1. Exceptions

Exception mechanisms in programming languages identify and handle undefined or abnormal arguments of a procedure, or elements of a class. We are here concerned only with the identification of exceptions, which is a form of incremental modification, and not with exception handling, which involves language design issues beyond the scope of this paper.

Exceptions may enlarge a class by including exceptions in a class even though they do not satisfy the class specification or, alternatively, shrink a class by excluding exceptions even though they do satisfy the class specification. Exceptions that enlarge a class will be called additive exceptions while exceptions that shrink a class will be called subtractive exceptions, as in Figure 6.

Additive exceptions extend a class to elements that do not satisfy the constraints of class membership. For example, [Bo] has a real-estate class for houses priced under $200,000 that is extended to higher-priced houses by exceptions. [To] views birds as mammals that fly, and extends this class to "birds that can't fly" like ostriches and penguins by exceptions. The base class is extended by admitting classes or individual elements that violate the class constraint.

Cancellation of an attribute may be viewed as the limiting case of broadening a constraint so it is always satisfied. Finer granularity for subclass extension is obtained by allowing constraints to be modified rather than altogether eliminated. An operational mechanism for handling additive exceptions by relaxation of constraints is developed in [Bo] for both exception subclasses and exception instances.

Subtractive exceptions define new classes by exclusion of classes (or individual elements) from a base class. For example, the class Voter may be defined from the base class Person by a variety of exclusionary exception classes like minors, felons etc. The class Retiree could be defined from the base class Person by an subtractive exception class Nonretirees.

Class exceptions provide a general framework for modeling incremental modification. Additive and subtractive incremental modification may be defined in terms of additive and subtractive class exceptions. Traditional inheritance has emphasized subtractive incremental modification while class exceptions have emphasized additive incremental modification. However, our perspective allows us to identify these two mechanisms for the management of change.

Mechanisms for additive and subtractive incremental modification of sets arise in other disciplines. For example, type-1 and type-2 errors in statistics, corresponding to errors of omission and commission, can be modeled by additive and subtractive exceptions. In logic, incompleteness may be mitigated by additive exceptions for true but unprovable formulae, while unsoundness may be mitigated by subtractive exceptions for provable but untrue formulae.

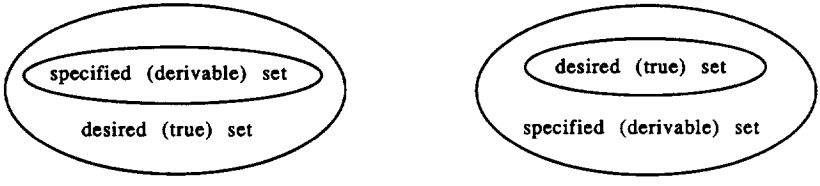
Figure 6 may be interpreted in the domain of logic. The formulae of a formal system have interpretations in which theorems denote objects in a possible world. The axioms determine a class specification and the theorems derivable from the axioms determine the elements of the class. Exceptions arise when the set of provable (valid) formulae do not correspond to the set of derivable (true) formulae. When the derivable formulae are a subset of the true formulae (incompleteness) additive exceptions may be used to augment the set of valid formulae. For example, the class of all birds is obtained from flying birds by augmenting it with the class of non-flying birds. When not all derivable formulae are true (unsoundness) subtractive exceptions may be used to eliminate provable untrue formulae. For example, the class of voters is obtained from the class of persons by eliminating minors and felons.

Cancellation of attributes may be used for specifying exceptions but is by itself a relatively inflexible mechanism. Adjusting constraints on given attributes by either weakening or strengthening them is more flexible (has finer granularity). Weakening of attribute constraints enlarges the associated class and is realized by additive exceptions, while strengthening attribute constraints determines a subclass and is realized by subtractive exceptions.

5. Abstract Inheritance and Nonmonotonicity

5.1. Inheritance as an Abstract Relation

We examine the relation between programming language inheritance and the abstract concept of inheritance developed by Brachman [Br1, Br2], Touretzky [To], Etherington [Et],



6a: Additive exceptions

6b: Subtractive exceptions

Figure 6: Additive and Subtractive Exceptions

and Reiter [Re] for semantic networks.

Programming language inheritance is an incremental modification operation on record structures with overlapping inheritable attributes. It is an imperative mechanism for creating new classes and associated objects during system evolution.

In contrast, abstract inheritance is a declarative relation among abstract entities in an inheritance network. Nodes of an inheritance network have no internal structure. The only structure is that determined by edges between neighboring nodes. Inheritance allows global relations among non-neighbors to be inferred from local relations among neighbors.

The simplest inheritance networks are tree structures with just a single kind of edge called an is-a relation. The properties of inheritance in such a network are simply the axioms of partial ordering, namely reflexivity, antisymmetry and transitivity. They allow inferences about inheritance between ancestors and descendants to be inferred from knowledge of inheritance between parents and children.

Bipolar inheritance networks with two kinds of links, called is-a and is-not links, are more expressive than unipolar nets. The semantics of bipolar nets may be characterized by the following axioms, where p, q, r represent classes, a represents instances, and x represents classes or instances [THT1]:

reflexivity of is-a: p is-a p
transitivity of is-a: p is-a q and q is-a r implies p is-a r
symmetry of is-not: p is-not q implies q is-not p
backward is-not propagation: x is-a q and q is-not r implies x is-not r
is-not inference for instances: p is-a q and a is-not q implies a is-not p

Although nodes have no internal structure, there are two kinds of nodes (class and instance nodes). "a is-a p" represents class membership (set membership) while "p is-a q" represent a subclass (subset) relation.

Bipolar semantic nets have been used to model multiple inheritance. However, when different paths of a multiple inheritance net yield contradictory conclusions more subtle analysis is required [THT2]. For example, a semantic net with the following links yields a contradiction:

Nixon is-a Quaker
Nixon is a Republican
Forall(x) Quaker(x) is-a Pacifist(x)
  Quakers are pacifists
Forall(x) Republican(x) is-not Pacifist(x)
  Republicans are not pacifists

The doubt concerning whether Nixon is a pacifist is resolved in different ways by skeptical reasoners who give up easily, refusing to reason further about apparently contradictory conclusions, and credulous reasoners, who examine further consequences of both sides of a contradiction.

The apparent contradiction may be resolved by allowing classes to have exceptions. Nixon is clearly either a nonpacifist Quaker or a pacifist Republican. There is no way of deciding between these two alternatives on the basis of the given information, but we happen to know Nixon is a nonpacifist Quaker, and therefore an exception to the rule that Quakers are pacifists.

Exceptions may be handled by default logic [Re], which allows default reasoning that may have to be revised as more information becomes available. Default reasoning is nonmonotonic since adding new facts may cause previously valid inferences to become invalid.

5.2. Nonmonotonicity

Monotonicity is a desirable property of incremental modification mechanisms that guarantees preservation of properties of a parent in an incrementally modified result. It arises in many different contexts.

Monotonic functions are functions over an ordered domain that preserve ordering of arguments, guaranteeing that larger arguments can never lead to a smaller result. Scott [Sc] interprets partial ordering over arguments that are functions in terms of their degree of definition, so that monotonicity becomes the property that a more defined function can never be transformed into a less defined result.

Monotonic inference is inference in which already proved theorems and facts can never become untrue. Classical logics are monotonic. Inference systems for abstract inheritance with exceptions are nonmonotonic because general (default) inferences about inheritance may later be negated by information about specific exceptions. For example, the general inference that Nixon is a pacifist because he is a Quaker must be revised when the specific information that Nixon is not a pacifist is added to the database. The general inference that ostriches are not birds because they do not fly is negated by the specific information that ostriches are non-flying birds.

In the context of inheritance, monotonicity is the condition that properties of a parent class are preserved in the result class. Our discussion of behavioral compatibility indicated that this condition was too strong for incremental software evolution. All forms of incremental modification other than complete behavioral compatibility are in fact nonmonotonic.

Nonmonotonic systems are untidy because we cannot assume that properties of classes or validity of inferences are preserved once they have been established. But in spite of this they are necessary because incremental system evolution is in practice nonmonotonic.

5.3. Structural Versus Abstract Inheritance

The emphasis in the present paper is on structural inheritance among entities whose internal structure consists of finite sets of attributes. From the abstract point of view, this corresponds to a particular model (interpretation) of abstract inheritance that imparts an attribute structure to abstract entities. Frames are another concrete model of abstract inheritance that is in many respects similar to the object-oriented interpretation.

Our concrete model of inheritance allows us to develop algorithms for implementing inheritance that depend on the specialized structure of the model. Our structural relations go beyond is-a and is-not relations to include a variety of incremental modification mechanisms which can be defined in a variety of interesting and useful ways in terms of the structure of inherited and inheriting entities. However, the abstract point of view complements the concrete structural view in exhibiting the fundamental abstract structure of inheritance and allowing us to see clearly the differences between inheritance in abstract structures and concrete computational structures.

Programming language inheritance hierarchies have is-a relations but have not felt a need for is-not relations, although such a relation could easily be defined operationally. Is-not is a potentially useful relation. For example, if we have "Chevy is-a Car" and "Toyota is-a Car", then the relation "Chevy is-not Toyota" tells us the important fact that Chevy and Toyota are disjoint sets. Inheritance hierarchies make no provision for expressing such information because they are viewed as imperative structure for finding inherited operations

rather than declarative structures for expressing relations. The integration of declarative and imperative features of inheritance hierarchies deserves further study.

The nodes of a bipolar net may be interpreted as propositions with "x is-a p" asserting p(x) and "x is-not p" asserting not(p(x)). In the domain of programming languages nodes are interpreted as types. The correspondence between the interpretation of nodes as propositions and types has an analogue in the domain of constructive type theory [ML]. In the present context, just as in constructive type theory, the interpretation of entities as propositions is simply an abstraction of their interpretation as types. The basis for the abstraction arises from the strong relation between monotonic inference and behavior-preserving type inheritance.

6. Parametric (Generic) Types

Parametric types capture a form of type similarity fundamentally different from the incremental similarity captured by inheritance. Whereas inheritance allows adding, deleting, and modifying operations, parametric types have a fixed set of generic (polymorphic) operations that may be specialized in a uniform manner, often allowing the same code to be used for a wide range of specializations. The similarity among parametric types is called "parametric similarity" in contrast to the "incremental similarity" determined by inheritance.

We are here interested primarily in incremental similarity but examine its relation to parametric similarity because it is better understood and has been widely studied in the context of ML [Mi], CLU [LSAS], and Ada [DOD]. The differences between incremental and parametric similarity have been studied in [Me] and [CW] and are illustrated by the following comparison:

(1) The dynamic binding of "self" to different objects and types at execution time has no counterpart for generic types. Instantiation of a generic type to a particular non-generic type occurs at compile time, while specialization of an inherited type occurs at execution time by binding "self" to an invoking object. Generic types are conceptually instantiated at compile time by macro expansion or an equivalent technique, while inherited types are shared and may be modified by "blocking" or "method combination" at execution time

(2) Generically similar types specialize a fixed set of polymorphic operations, while incrementally similar types add, cancel, or modify operations. Parametric and incremental similarity are complementary mechanisms for type modification that play different roles in application programming.

(3) Parametric types must be instantiated to specific types before they can be used as templates for creating objects, and instantiated types cannot in general be further instantiated. Inherited types can generally serve as both ancestors for inheritance by descendant types and templates for instances, and incrementally defined types in an inheritance hierarchy can generally be further incremented.

The ability to add new operations may be expressed by an implicit formal parameter that may optionally be instantiated to an arbitrary descendant type that in turn has an implicit formal parameter for further incremental modification. "Self" is an additional implicit parameter that is instantiated at operation invocation time to the object on whose behalf an operation is invoked.

Thus parametric types have an explicit formal parameter that parameterizes a fixed set of parametric operations, while inherited types have an implicit parameter that serves to augment the set of operations when it is instantiated by a subtype in the context of an

invocation of an object of the subtype. These two forms of parameterization represent complementary forms of variability. In particular, parametric types are useful in representing common structure and behavior for a range of types, such as that of stacks or lists of arbitrary type, while inherited types are useful in incrementally augmenting the set of operations for the purpose of specialization or evolution.

Generative similarity is useful for generating multiple instances of similar objects whose differences are captured by a parameter. Incremental similarity is useful in managing change that arises during evolution or exception handling. Generative similarity and the associated mechanism of parametric types are more uniform and have more developed mathematical models than incremental similarity. The relation between generic and incremental similarity has been studied in [Me] and [CW].

Parametric similarity was first studied by Strachey [Str]' and was the basis for the development of ML [Mi, Ha]. We view parametric polymorphism as a particular kind of type similarity. Looking at polymorphism in this way in turn suggests that other forms of type similarity may correspond to other forms of polymorphism. For example signature-compatible type similarity has been called "inclusion polymorphism" [CW]. Inheritance determines a more permissive form of polymorphism that we call "incremental polymorphism". Incremental polymorphism includes inclusion polymorphism (signature compatibility) as a special case.

Signature compatibility was examined in [Ca]. A framework for combining parametric polymorphism, signature compatibility, and data abstraction was developed in [CW]. Incremental similarity is more general than signature compatibility in that it captures behavioral as well as domain-range similarity, and forms of incremental change that violate signature compatibility in order to model exceptions and system evolution. Such generalization requires sacrificing static type checking but permits more adequate modeling of evolution in real-world applications.

Figure 7 illustrates the similarities and differences between parametric and incremental notions of type similarity. In both cases a common ancestor captures the similarity of a collection of descendants. But edges connecting ancestors to their descendants have an entirely different meaning in these two cases. For parametric types, edges connect a generic type to an instantiation for a particular parameter value. For inheritance hierarchies, edges connect ancestor types to descendants that incrementally modify the set of operations of the ancestor.

The relation between parametric and incremental similarity may be illustrated by thinking of operations of a type as instruments in an orchestra. Variations of a particular melody can be realized by changing the pitch and volume of a fixed set of instruments or by adding harmonies for new instruments. Variations with a fixed set of instruments correspond to

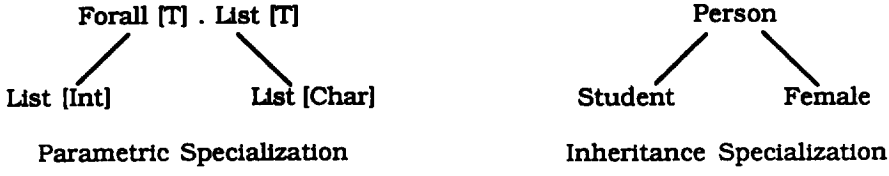


Figure 7: Common Ancestors Versus Parametric Types

parametric similarity while harmonies with new instruments correspond to incremental similarity. Mozart was able to achieve new musical effects by making use of the crescendo, a new form of musical technology that extended the range of parametric variation. Inheritance is an example of new computational technology that extends the range of incremental variation.

7. Liketypes

The four incremental modification mechanisms may be named as follows:

behavioral compatibility: R subtype P or R is-a P
signature compatibility: R subsig P
name compatibility: R subclass P
cancellation: R1 like R2

Like is the most general of the relations and subsumes all the others. Types related by a like relation may be called *liketypes* by analogy with the term subtypes for types related by a subset relation.

Whereas the subtype relation is asymmetrical, the like relation is symmetrical:

if "R1 like R2" then "R2 like R1".

Cancellation may be combined with the is-a relation to obtain a symmetrical relation strictly weaker than the general like relation which we call is-like [We2]:

R1 is-like R2 iff ((Exists P) [R1 is-a P and R2 is-a P])

This relation holds between common subtypes of a parent P. It is useful in modeling type evolution from a common parent. When we are given just the subtypes R1 and R2 then the problem of determination of the common ancestor P may be formulated as a unification problem [AN].

Since only is-a relations may obtain between components of R1 and R2 and their common ancestor P, the relation is-like is strictly weaker than the general like relation. Corresponding symmetric relations for signature and name compatibility may be defined:

R1 sig-like R2 iff ((Exists P) [R1 subsig P and R2 subsig P])
R1 name-like R2 iff ((Exists P) [R1 subclass P and R2 subclass P])

In many applications we are interested in formulating the relation between similar liketypes in terms of their differences rather than their similarities. For example, consider a military application where the type "tank" evolves through a sequence of small changes to its armor. Let T1, T2, T3 be three successive versions of the type "tank". Then T2 can be defined by a minor modification of T1 and T3 by a minor modification of type T2. We have "T1 like T2" and "T2 like T3", where the common properties C12 of T1 and T2 and C23 of T2 and T3 include practically all the properties of tanks, but are likely to be slightly different.

In these circumstances incremental evolutionary changes M12, M23 are more descriptive than common properties C12, C23:

T2 = T1 + M12
T3 = T2 + M23

The relation "R1 like R2" may be interpreted as an assertion that is true or false or as an incremental modification rule for specifying R2 = R1 + M in terms of R1. For purposes of application programming, the interpretation of "R1 like R2" as a rule for incrementally specifying R2 in terms of R1 is often more useful. Specification of R2 in terms of differences from R1 is often preferable to specification in terms of common attributes C.

8. Combining Like Relationships

Current languages generally select a single incremental subtyping mechanism that is built into its type system. However, each incremental type definition technique is methodologically useful. We therefore consider combining them all in a single programming language.

Such a language would have the ability to accept subtype definitions that establish any of the like relationships discussed here between a type T and a descendant type S. The type lattice could have freely mixed inheritance links representing any of the varieties of type similarity.

It is important to understand how such a heterogeneous type lattice affects our type-checking mechanism. We will do this by looking at the transitivity properties of like relationships. Consider first the interaction between is-a and name-like (abbreviated in Figure 8 to like) inheritance.

Suppose that "B name-like A", since it redefines the f method defined in A to be f'. This notation indicates that f and f' have the same name, but their bodies are different. Also, "C name-like B", since it redefines the g method that was introduced in B, as in Figure 8(a). The result is that C name-like A, since C inherits the modified f method from B.

Suppose that B name-like A by rewriting the f method, and C name-like B in that it adds one new method g. This is the case shown in Figure 8(b). Similar to the case above, A and C are related as C name-like A, since C inherits the redefined f method from B.

Suppose that B is-a A by adding one method and C name-like B in that it has the same methods but has redefined one of them, as in Figure 8(c). Thus, C is-a A because none of the methods on A have been modified in C.

From this we see that in order to determine the transitive relationship between A and C in situations like the above, we need to analyze the relationship between corresponding methods at each level of the lattice.

In the example below the four kinds of liketypes are abbreviated as in section 7. We also assume the function name(f) returns the name of the function f, and the function sig(f)

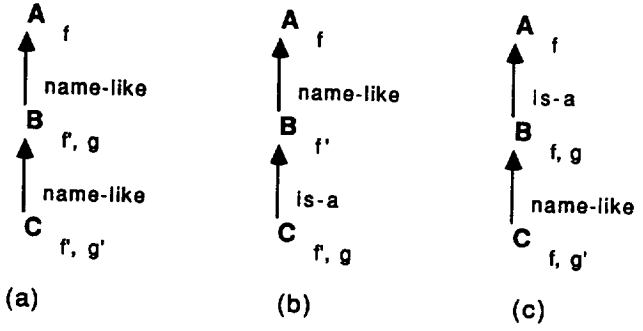


Figure 8. Transitive Like Relationships.

returns the signature of f.

The general rules for reasoning about the relationship between two types in the type lattice that are on a common path to the root can be stated as follows. If A supports the methods f_{a1},..., f_{am}, B supports the methods f_{b1},..., f_{bn}, and B is an indirect subtype of A, then

If [For all f_{ai}] [There exists f_{bj}] ($f_{ai} = f_{bj}$) **then** B **is-a** A
else if [For all f_{ai}] [There exists f_{bj}] (sig (f_{ai}) = sig (f_{bj})) **then** B **sig-like** A
else if [For all f_{ai}] [There exists f_{bj}] (name (f_{ai}) = name (f_{bj})) **then** B **name-like** A
else B **like** A

With this kind of analysis, we can determine how to handle type checking in our programming language. For example, consider the following piece of code:

```
x: A;
y: C;
x := y;
```

where C is an indirect subtype of A. In order to determine if the assignment is allowable, we can first apply the above rule to the types A and C to determine the effective relationship between them. We can then use the rules of the language for type compatibility between a type and its immediate subtypes to determine whether or not a particular assignment is legal.

Consider the example shown in Figure 9. The effective relationship between Vehicle and Toyota is name-like. If our language does not guarantee behavioral compatibility between types (as in Smalltalk), then the first assignment would be legal. Toyota supports all of the named behavior of Vehicle, allowing the substitution of a Toyota wherever a Vehicle was expected. It will always be possible to drive either, even though they might have different semantics.

The second assignment, however, would not be allowed in a language that guaranteed the kind of substitution mentioned in the previous paragraph. The rules tell us that the relationship between Vehicle and Car-Sculpture is a like. This means that we cannot use a Car-Sculpture wherever a Vehicle was expected, because someone might try to drive it.

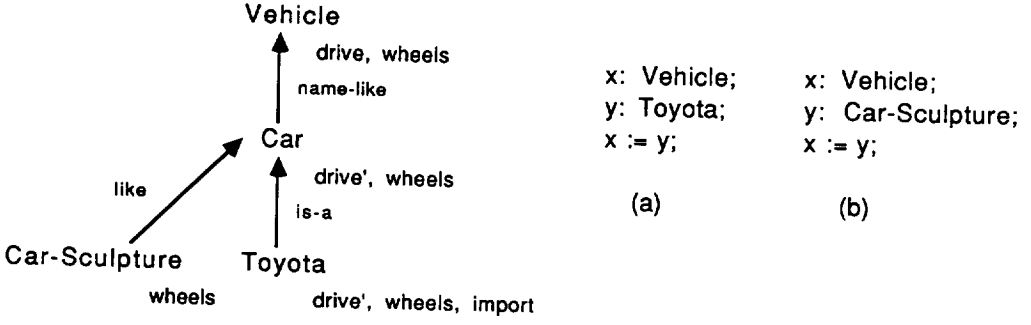


Figure 9. Example of type checking

For type checking on function application, we must simply match the compatibility rules for that function. That is, if we have a function f defined on type A and we try to apply f to a member of an indirect subtype B, we must ensure that the definition of f available on that subtype meets the compatibility rules of the language. If the language requires behavioral compatibility, then we must check that no type between A and B redefines f. Notice that the effective relationship between A and B might be a weak form of like; however, this is irrelevant since it is only the one function f that matters.

9. Conclusion

This paper contributes to the design of inheritance mechanisms by presenting major design alternatives and exploring the models, motivation, and methodology underlying each alternative.

We started with the objective of better understanding inheritance by classifying the varieties of incremental modification that are legitimate realizations of inheritance. This led us to a discussion of the essence of inheritance and of the distinction between types and classes. It led to a novel algebraic characterization of alternative forms of behavioral compatibility, to the analysis of horizontal and vertical signature compatibility, and to the distinction between additive and subtractive exceptions as a model of additive and subtractive incremental modification. It led to the distinction between concrete and abstract inheritance and between monotonic and nonmonotonic incremental modification. It led to an exploration of how multiple like mechanisms can be combined in a single language. The fact that this approach has yielded many interesting analyses suggests that viewing inheritance as an incremental modification mechanism is worthwhile.

10. Acknowledgements

The authors are indebted to William Cook for contributions to the section on inheritance and for the term "liketype", to Lynn Stein for contributions to the section on types, to Kim Bruce for contributions to the algebraic specification of behavioral compatibility, and to numerous other readers who have made helpful comments. This work was supported in part by NSF under contract DCR-8605567, by DARPA under order #4768, and by the IBM TJ Watson Research Center.

11. References

[AN] Ait Kaci H. and Nasr R., Login: A Logic Programming Language with Built-in Inheritance, Journal of Logic Programming, 1986.

[Bo] Borgida A., Exceptions in Object-Oriented Languages, Sigplan Notices, Oct 1986.

[Br1] Brachman, R., "I Lied About the Trees Or Defaults and Definitions in Knowledge Representation", AI Magazine, Fall, 1985.

[Br2] Brachman, R., "What Is-a is and Isn't", AI Magazine, Fall 1985.

[BW] Bruce K. B., and Wegner P., An Algebraic Model of Subtypes and Inheritance, Brown University Report, July 1987, also in Proc Roscoff Conference on Database Programming Languages, Sept 1987.

[Ca] Cardelli L., A Semantics of Multiple Inheritance, In LNCS Vol 173, Ed G. Kahn, 1984.

[CW] Cardelli, L. and Wegner, P., On Understanding, Types, Data Abstraction, and Polymorphism, ACM Computing Surveys, December, 1985.

[Co1] Cook, W., A Denotational Semantic Model of Inheritance, Forthcoming PhD Thesis, Brown University Summer 1988.

[Co2] Cook, W., The Semantics of Inheritance, Brown University Technical Report, March 1988.

[DOD] Ada Reference Manual, US Dept of Defense, July 1980.

[Et] Etherington D. W., Formalizing Nonmonotonic Reasoning, Artificial Intelligence, 1987.

[GR] Goldberg, A. and Robson, Smalltalk80: The Language and Its Implementation, Addison-Wesley, 1983.

[Ha] Harper R., An Introduction to ML, Edinburgh University Technical Report, 1987.

[LSAS] Liskov B., Snyder A., Atkinson R., and Schaffert C., Abstraction Mechanisms in CLU, CACM, August 1987.

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

[Mi] Milner R., A Proposal for Standard ML, Proc Symposium on Lisp and Functional Programming, ACM, August 1984.

[ML] Martin-Lof P., Constructive Mathematics and Computer Programming, in Mathematical Logic and Computer Programming, Hoare and Shepherdson Eds, Prentice Hall International, 1985.

[Re] Reiter R., A Logic for Default Reasoning, Artificial Intelligence, 1980.

[Sc] Scott D., Data Types as Lattices, Siam Journal of Computing, September 1976.

[Sn] Snyder, A., Encapsulation and Inheritance in Object-Oriented Languages, OOPSLA, 1986.

[Str] Strachey C., Fundamental Concepts in Programming Languages, lecture Notes for International Summer School in Computer Programming, Copenhagen, August 1967.

[THT1] Thomason, R. H., Horty, J. F., and Touretzky, D., A Calculus for Inheritance in Monotonic Semantic Nets, CMU-CS-86-138, July 1986.

[THT2] Touretzky, D. S., Horty, J. F., and Thomasson R. H., A Clash of Intuitions: The Current State of Nonmonotonic Multiple Inheritance Systems, Proc IJCAI-87.

[To] Touretzky, D., The Mathematical Theory of Inheritance, Morgan-Kaufman, 1986.

[We1] Wegner, P., Object-Oriented Concept Hierarchies, Brown University Technical Report, May 1988.

[We2] Wegner, P., The Object-Oriented Classification Paradigm, in Research Directions in Object-Oriented Programming, Edited by Shriver and Wegner, MIT Press, 1987.

[Zd] Zdonik S. B., Can Objects Change Type? Can Type Objects Change?, Proc Roscoff Conference on Database Programming Languages, Sept 1987.