A New Definition of the Subtype Relation

Barbara Liskov[1] and Jeannette M. Wing[2]

[1] MIT Laboratory for Computer Science
Cambridge, MA 02139, USA
[2] School of Computer Science, Carnegie Mellon University,
Pittsburgh, PA 15213, USA

Abstract. The use of hierarchy is an important component of object-oriented design. Hierarchy allows the use of type families, in which higher level supertypes capture the behavior that all of their subtypes have in common. For this methodology to be effective, it is necessary to have a clear understanding of how subtypes and supertypes are related. This paper presents a new definition of the subtype relation that ensures that any property proved about supertype objects also holds for subtype objects. It also discusses the ramifications of the definition on the design of type families.

1 Introduction

What does it mean for one type to be a subtype of another? We argue that this is a semantic question having to do with the relationship between the specifications of the two types. In this paper we give a precise definition of the subtype relation in terms of the behavior of types as described by their specifications. Our definition extends earlier work by providing for subtypes that have more methods than their supertypes, and by allowing for sharing of mutable objects among multiple users. We also discuss the ramifications of the definition with respect to various kinds of subtype relationships and give examples of type families that satisfy the definition.

To motivate our notion of subtyping, consider how subtypes are used in object-oriented programming languages. In strongly typed languages such as Simula 67, C++, Modula-3, and Trellis/Owl, subtypes are used to broaden the assignment statement. An assignment

x: T := E

is considered to be legal provided the type of expression E is a subtype of the declared type T of variable x. Once the assignment has occurred, x will be used according to its "apparent" type T, with the expectation that if the program performs correctly when the actual type of x's object is T, it will also work correctly if the actual type of the object denoted by x is a subtype of T.

Clearly subtypes must provide the expected methods with compatible signatures. This consideration has led to the formulation by Cardelli of the contra/covariance rules [5]. However, these contra/covariance rules are not strong

enough to ensure that the program containing the above assignment will work correctly for any subtype of T, since all they do is ensure that no type errors will occur. It is well known that type checking, while very useful, captures only a small part of what it means for a program to be correct; the same is true for the contra/covariance rules.

For example, consider stacks and queues. These types might both have a *put* method to add an element and a *get* method to remove one. According to the contravariance rule, either could be a legal subtype of the other. However, a program written in the expectation that x is a stack is unlikely to work correctly if x actually denotes a queue, and vice versa.

What is needed is a stronger requirement that constrains the behavior of subtypes: the subtype's objects must behave "the same" as the supertype's as far as anyone using the supertype's objects can tell. This paper is concerned with obtaining a precise definition of this "subtype requirement." Our definition is applicable to a particularly general environment, one that allows multiple, possibly concurrent, users to share mutable objects; the environment is discussed further in Section 2. Although the states of objects in such an environment may reflect changes due to the activities of several users, we still want individual users to be able to make deductions about the current states of objects based on what they observed in the past. These deductions should be valid if they follow from the specification of an object's presumed type even though the object is actually a member of a subtype of that type and even though other users may be manipulating it using methods that do not exist for objects of the supertype.

In other words, we want the subtype to preserve *safety* properties ("nothing bad happens") that hold for the supertype. There are two kinds of safety properties: *invariant* properties, which are properties true of all states, and *history* properties, which are properties true of all sequences of states. For example, for a stack, an invariant property we might want to prove is that its size is always greater or equal to zero; a history property is that its bound never changes. We might also want to prove *liveness* properties ("something good eventually happens"), e.g., an element pushed onto a stack will eventually be popped, but our focus here will be just on safety properties. Our definition of subtype will guarantee that all the invariant and history properties that hold for objects of the supertype also hold for objects of the subtype.

Our approach lets programmers reason directly in terms of the specifications rather than the underlying mathematical models of types, be they algebras, categories, or higher-order lambda expressions. Our definition is motivated by pragmatic concerns: we wanted to make our ideas accessible to everyday programmers. We provide a simple checklist that can be used by programmers in a straightforward way to validate a proposed design of a type hierarchy. In this paper, we use informal specifications; see [20] for formal ones.

This paper makes two important technical contributions:

1. It provides a very general yet easy to use definition of the subtype relation. Our definition extends earlier work, including the most closely related work done by America [3], by allowing subtypes to have more methods than their

supertypes.

2. It discusses the ramifications of the subtype relation and shows how interesting type families can be defined. For example, arrays are not a subtype of sequences (because the user of a sequence expects it not to change over time) and 32-bit integers are not a subtype of 64-bit integers (because a user of 64-bit integers would expect certain method calls to succeed that will fail when applied to 32-bit integers). We show in Section 4 how useful type hierarchies that have the desired characteristics can be defined.

The paper is organized as follows. We describe our model of computation in Section 2. In Section 3 we present and discuss our formal definition of subtyping, motivating it informally with an example relating stacks to bags. Section 4 discusses the ramifications of our definition on designing type hierarchies. We describe related work in Section 5, and then close with a summary of contributions.

2 Model of Computation

We assume a set of all potentially existing objects, *Obj*, partitioned into disjoint typed sets. Each object has a unique identity. A *type* defines a set of *legal values* for an object and a set of *methods* that provide the only means to manipulate that object. An object's actual representation is encapsulated by its set of methods.

Objects can be created and manipulated in the course of program execution. A *state* defines a value for each existing object. It is a pair of two mappings, an *environment* and a *store*. An environment maps program variables to objects; a store maps objects to values.

$$State = Env \times Store$$
$$Env = Var \rightarrow Obj$$
$$Store = Obj \rightarrow Val$$

Given an object, x, and a state ρ with an environment, e, and store, s, we use the notation x_ρ to denote the value of x in state ρ; i.e., $x_\rho = \rho.\mathrm{s}(\rho.\mathrm{e}(\mathrm{x}))$. When we refer to the domain of a state, $\mathrm{dom}(\rho)$, we mean more precisely the domain of the store in that state.

We model a type as a triple, $< O, V, M >$, where $\mathrm{O} \subseteq Obj$ is a set of objects, $\mathrm{V} \subseteq Val$ is a set of legal values, and M is a set of methods. Each method for an object is a *constructor*, an *observer*, or a *mutator*. Constructors of an object of type τ return new objects of type τ; observers return results of other types; mutators modify the values of objects of type τ. A type is *mutable* if any of its methods is a mutator; otherwise it is *immutable*. We allow "mixed methods" where a constructor or an observer can also be a mutator. We also allow methods to signal exceptions; we assume termination exceptions, i.e., each method call either terminates normally or in one of a number of named exception conditions. To be consistent with object-oriented language notation, we write x.m(a) to denote the call of method m on object x with set of arguments a.

Objects come into existence and get their initial values through *creators*. Unlike other kinds of methods, creators do not belong to particular objects, but rather are independent operations. They are the "class methods"; the other methods are the "instance methods." (We are ignoring other kinds of class methods in this paper.)

A *computation*, i.e., program execution, is an alternating sequence of states and statements starting in some initial state, ρ_0:

$$\rho_0 \;\; S_1 \;\; \rho_1 \;\; \cdots \;\; \rho_{n-1} \;\; S_n \;\; \rho_n$$

Each statement, S_i, of a computation sequence is a partial function on states. A *history* is the subsequence of states of a computation. A state can change over time in only three ways[3]: the environment can change through assignment; the store can change through the invocation of a mutator; the domain can change through the invocation of a creator or constructor. We assume the execution of each statement is atomic. Objects are never destroyed:

$$\forall\, 1 \leq i \leq n \;.\; dom(\rho_{i-1}) \;\; \subseteq \;\; dom(\rho_i).$$

Computations take place within a universe of shared, possibly persistent objects. Sharing can occur not only within a single program through aliasing, but also through multiple users accessing the same object through their separate programs. We assume the use of the usual mechanisms, e.g., locking, for synchronizing concurrent access to objects; we require that the environment uses these mechanisms to ensure the atomicity of the execution of each method invocation. We are interested in persistence because we imagine scenarios in which a user might create and manipulate a set of objects today and store them away in a persistent repository for future use, either by that user or some other user. In terms of database jargon, we are interested in concurrent transactions, where we are ignoring aborts and the need for recovery. The focus of this paper is on subtyping, not concurrency or recoverability; specific solutions to those problems should apply in our context as well.

3 The Meaning of Subtype

3.1 The Basic Idea

To motivate the basic idea behind our notion of subtyping, let's look at a simple-minded, slightly contrived example. Consider a bounded bag type that provides *put* and *get* methods that insert and delete elements into a bag. *Put* has a pre-condition that checks to see that adding an element will not grow the bag beyond its bound. *Get* has a pre-condition that checks to see that the bag is non-empty. Informal specifications [19] for *put* and *get* for a bag object, b, are as follows:

[3] This model is based on CLU semantics.

put = **proc** (i: int)
requires The size of b is less than its bound.
modifies b
ensures Inserts i into b.

get = **proc** () **returns** (int)
requires b is not empty.
modifies b
ensures Removes and returns some integer from b.

Here the **requires** clause states the pre-condition. The **modifies** and **ensures** clauses together define the post-condition; the **modifies** clause lists objects that might be modified by the call and thus indicates that objects not listed are not modified.

Consider also a bounded stack type that has, in addition to *push* and *pop* methods, a *swap_top* method that takes an integer, i, and modifies the stack by replacing its top with i. Stack's *push* and *pop* methods have pre-conditions similar to bag's *put* and *get* and *swap_top* has a pre-condition requiring that the stack is non-empty. Informal specifications for methods of a stack, s, are as follows:

push = **proc** (i: int)
requires The height of s is less than its bound.
modifies s
ensures Pushes i onto the top of s.

pop = **proc** () **returns** (int)
requires s is not empty.
modifies s
ensures Removes the top element of s and returns it.

swap_top = **proc** (i: int)
requires s is not empty.
modifies s
ensures Replaces s's top element with i.

Intuitively, stack is a subtype of bag because both are collections that retain an element added by *put*/*push* until it is removed by *get*/*pop*. The *get* method for bags does not specify precisely what element is removed; the *pop* method for stack is more constrained, but what it does is one of the permitted behaviors for bag's *get* method. Let's ignore *swap_top* for the moment.

Suppose we want to show stack is a subtype of bag. We need to relate the values of stacks to those of bags. This can be done by means of an *abstraction function*, like that used for proving the correctness of implementations [13]. A given stack value maps to a bag value where we abstract from the insertion order on the elements.

We also need to relate stack's methods to bag's. Clearly there is a correspondence between the stack's *put* method and bag's *push* and similarly for the *get* and *pop* methods (even though the names of the corresponding methods do not match). The pre- and post-conditions of corresponding methods will need to relate in some precise (to be defined) way. In showing this relationship we need to appeal to the abstraction function so that we can reason about stack values in terms of their corresponding bag values.

Finally, what about *swap_top*? There is no corresponding bag method so there is nothing to map it to. However, intuitively *swap_top* does not give us any additional computational power; it does not cause a modification to stacks that could not have been done in its absence. In fact, *swap_top* is a method on stacks whose behavior can be explained completely in terms of existing methods. In particular,

s.swap_top(i) = s.pop(); s.push(i)

If we have a bag object and know it, we would never call *swap_top* since it is defined only for stacks. If we have a stack object, we could call *swap_top*; but then for stack to still be a subtype of bag, we need a way to explain its behavior so that a user of the object as a bag does not observe non-bag-like behavior. We use an *extension map* for this explanation. We call it an extension map because we need to define it only for new methods introduced by the subtype.

The extension map for *swap_top* describes what looks like a straight-line program. A more complicated program would be required if stack also had a method to *clear* a stack object of all elements:

clear = **proc** ( )
modifies s
ensures Empties s.

This method would be mapped to a program that repeatedly used the *pop* method to remove elements from the stack until its size is zero (assuming there is a way to determine the size of a bag and stack, e.g., a more realistic specification of *get* would signal an exception if passed an empty bag).

Here then is our basic idea: Given two types, σ and τ, we want to say that σ is a subtype of τ if there exist correspondences between their respective sets of values and methods. Relating values is straightforward; we use an abstraction function. Relating methods is the more interesting part of our notion of subtyping. There are two main ideas. Informally, we require that:

- σ must have a corresponding method for each τ method. σ's corresponding method must have "compatible" behavior to τ's in a sense similar to its signature being "compatible" according to the usual contra/covariance rules. This boils down to showing that the pre-condition of τ's method implies that of σ's and the post-condition of σ's implies that of τ's. (We will see later that our actual definition of subtyping is slightly weaker.)
- If σ adds methods that have no correspondence to those in τ, we need a way to explain these new methods. So, for each new method added by σ to τ, we

need to show "a way" that the behavior of the new method could be effected by just those methods already defined for τ. This "way" in general might be a program.

3.2 Formal Definition

Our definition relies on the existence of specifications of types. The definition is deliberately independent of any particular specification language, but we do require that the specification of a type $\tau = < O, T, N >$ contain the following information:

- A description of the set of *legal* values, T.
- A description of each method, m $\in$ N, including:
  - its signature, i.e., the number and types of the arguments, the type of the result, and a list of exceptions.
  - its behavior, expressed in terms of a pre-condition, m.pre, and a post-condition, m.post. We assume these pre- and post-conditions are written as state predicates. We write m.pred for the predicate m.pre $\Rightarrow$ m.post.

The pre- and post-conditions allow us to talk about a method's side effects on mutable objects. In particular, they relate the final value of an object in a post state to its initial value in a pre state. In the presence of mutable types, it is crucial to distinguish between an object and its value as well as to distinguish between its initial and final values. We will use "pre" and "post" as the generic initial and final states in a method's specification. So, for example, x_{pre} stands for the value of the object x in the state upon method invocation.

To show that a subtype σ is related to supertype τ, we need to provide a *correspondence mapping*, which is a triple, $< A, R, E >$, of an *abstraction* function, a *renaming* function, and an *extension* mapping. The abstraction function relates the legal values of subtype objects to legal values of supertype objects, the renaming function relates subtype methods to supertype methods, and the extension mapping explains the effects of extra methods of the subtype that are not present in the supertype. We write $\sigma < \tau$ to denote that σ is a subtype of τ. Figure 1 presents our definition of the subtype relation.

Figure 2 illustrates the last clause of our definition, the diamond rule, which is key to handling extra mutators of the subtype. This diagram, read from top to bottom, is not quite like a standard commutative diagram because we are applying subtype methods to the same subtype object in both cases (m and E(x.m(a))) and then showing the two values obtained map via the abstraction function to the same supertype value.

3.3 Discussion

There are two kinds of properties that must hold for subtypes: (1) all calls to methods of the supertype have the same meaning when the actual call invokes a method of the subtype; (2) all invariants and history properties that hold for objects of the supertype must also hold for those of the subtype.

Definition of the subtype relation, $<$:
$\sigma = < O_\sigma, S, M >$ is a *subtype* of $\tau = < O_\tau, T, N >$ if there exists a correspondence mapping, $< A, R, E >$, where:

1. The abstraction function, $A : S \rightarrow T$, is total, need not be onto, but can be many-to-one.
2. The renaming map, $R : M \rightarrow N$, can be partial and must be onto and one-to-one. If m_τ of τ is the corresponding renamed method m_σ of σ, the following rules must hold:
   - *Signature rule.*
     - *Contravariance of arguments.* m_τ and m_σ have the same number of arguments. If the list of argument types of m_τ is α_i and that of m_σ is β_i, then $\forall i.\alpha_i < \beta_i$.
     - *Covariance of result.* Either both m_τ and m_σ have a result or neither has. If there is a result, let m_τ's result type be γ and m_σ's be δ. Then $\delta < \gamma$.
     - *Exception rule.* The exceptions signaled by m_σ are contained in the set of exceptions signaled by m_τ.
   - *Methods rule.* For all $x : \sigma$:
     - *Pre-condition rule.* $m_\tau.pre[A(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre$.
     - *Predicate rule.* $m_\sigma.pred \Rightarrow m_\tau.pred[A(x_{pre})/x_{pre}, A(x_{post})/x_{post}]$

     where P[a/b] stands for predicate P with every occurrence of b replaced by a. Since x is an object of type σ, its value (x_{pre} or x_{post}) is a member of S and therefore cannot be used directly in the pre- and post-conditions for τ's methods (which relate values in T). A is used to translate these values so that the pre- and post-conditions for τ's methods make sense.
3. The extension map, $E : O_\sigma \times M \times Obj* \rightarrow Prog$, must be defined for each method, m, not in *dom(R)*. We write E(x.m(a)) for E(x, m, a) where x is the object on which m is invoked and a is the (possibly empty) sequence of arguments to m. E's range is the set of programs, including the empty program denoted as ϵ.[4]
   - *Extension rule.* For each new method, m, of $x : \sigma$, the following conditions must hold for π, the program to which E(x.m(a)) maps:
     - The input to π is the sequence of objects $[x]||a$.
     - The set of methods invoked in π is contained in the union of the set of methods of all types other than σ and the set of methods *dom(R)*.
     - *Diamond rule.* We need to relate the abstracted values of x at the end of either calling just m or executing π. Let ρ_1 be the state in which both m is invoked and π starts. Assume m.pre holds in ρ_1 and the call to m terminates in state ρ_2. Then we require that π terminates in state ψ and
       $A(x_{\rho_2}) = A(x_\psi)$.
       Note that if $\pi = \epsilon, \psi = \rho_1$.

Fig. 1. Definition of the Subtype Relation

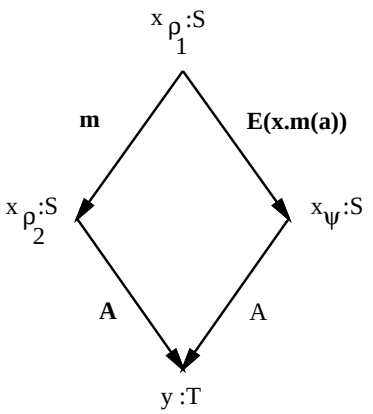

Fig. 2. The Diamond Diagram

The renaming map defines the correspondence between methods of the subtype and supertype. It allows renaming of the methods (e.g., the *put* method of bag can be renamed to *push*) because this ability is useful when there are multiple supertypes. For example, two types might use the same name for two different methods; without renaming it would be impossible to define a type that is a subtype of both of them.

The requirement about calls of individual methods of the supertype is satisfied by the signature and methods rules. The first two signature rules are the usual contra/covariance rules for "syntactic" subtyping as defined by Cardelli [5]; ours are adapted from America [3]. The exception rule says that m_σ may not signal more than m_τ since a caller of a method on a supertype object should not expect to handle an unknown exception. The pre-condition rule ensures the subtype's method can be called in any state required by the supertype as well as other states. The predicate rule when expanded is equivalent to:

$$(m_\sigma.pre \Rightarrow m_\sigma.post) \Rightarrow ((m_\tau.pre \Rightarrow m_\tau.post)[A(x_{pre})/x_{pre}, A(x_{post})/x_{post}])$$

which is implied by the stronger conjunction of the following separate pre- and post-condition rules that America uses:

Pre-condition rule. $m_\tau.pre[A(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre$
Post-condition rule. $m_\sigma.post \Rightarrow m_\tau.post[A(x_{pre})/x_{pre}, A(x_{post})/x_{post}]$

These two rules are the intuitive counterparts to the contravariant and covariant rules for signatures. The post-condition rule alone says that the subtype method's post-condition can be stronger than the supertype method's post-condition; hence, any property that can be proved based on the supertype method's post-condition also follows from the subtype's method's post-condition. Our pre-and-predicate rule differs from America's pre-and-post rule only when the subtype's method's pre-condition is satisfied and the supertype's method's

pre-condition is not. In this case we do not require that the post-condition of the subtype's method imply that of the supertype's method; this makes sense because specifications do not constrain what happens when a pre-condition is not satisfied. Our weaker formulation gives more freedom to the subtype's designer. (A similar formulation is used by Leavens [17].)

The requirement about invariants holding for values of objects of the supertype is satisfied by requiring that the abstraction function be defined on all legal values of the subtype and that each is mapped to some legal value of the supertype.

Preservation of history properties is ensured by a combination of the methods and extension rules; they together guarantee that any call of a subtype method can be explained in terms of calls of methods that are already defined for the supertype. Subtypes have two kinds of methods, those that also belong to the supertype (via renaming) and those that are "extra." The methods rule lets us reason about all the non-extra methods using the supertype specification. The extension rule explains the meaning of the extra methods in terms of the non-extra ones, thus relating them to the supertype specification as well. Note that interesting explanations are needed only for mutators; non-mutators always have the "empty" explanation, ϵ.

The extension rule constrains only what an explanation program does to its method's object, and not to other objects. This limitation is imposed because the explanation program does not really run. Its purpose is to explain how an object could be in a particular state. Its other arguments are hypothetical; they are not objects that actually exist in the object universe.

The diamond rule is stronger than necessary because it requires equality between abstract values. We need only the weaker notion of *observable equivalence* (e.g., see Kapur's definition [14]), since values that are distinct may not be observably different if the supertype's set of methods (in particular, observers) is too weak to let us perceive the difference. In practice, such types are rare and therefore we did not bother to provide the weaker definition.

3.4 Applying the Definition of Subtyping as a Checklist

Let's revisit the stack and bag example using our definition as a checklist. Here $\sigma = < O_{stack}, S, \{push, pop, swap_top\} >$, and $\tau = < O_{bag}, B, \{put, get\} >$. Suppose we represent a bounded bag's value as a pair, $< elems, bound >$, of a multiset of integers and a fixed bound, requiring that the size of the multiset, elems, is always less than or equal to the bound. E.g., $< \{7, 19, 7\}, 5 >$ is a legal value for bags but $< \{7, 19, 7\}, 2 >$ is not. Similarly, let's represent a bounded stack's value as a pair, $< items, limit >$, of a sequence of integers and a fixed bound, requiring that the length of items is always less than or equal to its limit. We use standard notation to denote functions on multisets and sequences.

The first thing to do is define the abstraction function, $A : S \rightarrow B$, such that for all st: S:

$$A(st) = < mk_elems(st.items), st.limit >$$

where the helping function, mk_elems, maps sequences to multisets. It is defined such that for all integer sequences, sq, and integers, i:

$$mk_elems([\,]) = \{\ \}$$
$$mk_elems(sq \,||\, [i]) = mk_elems(sq) \cup \{i\}$$

([ ] stands for the empty sequence and { } stands for the empty multiset; || is concatenation and $\cup$ is a multiset union operation that does not discard duplicates.)

Second, we define the renaming map, R:

R(push) = put
R(pop) = get

Checking the signature rule is easy and could be done by the compiler.

Next, we show the correspondences between *push* and *put*, and between *pop* and *get*. Let's look at the pre-condition and predicate rules for just one method, *push*. The pre-condition rule for *put*/*push* requires that we show:

The size of b is less than its bound. *Put*'s pre-condition.
⇒
The height of s is less than its bound. *Push*'s pre-condition.

or more formally[5],

$$size(A(s_{pre}).elems) < A(s_{pre}).bound$$
$$\Rightarrow$$
$$length(s_{pre}.items) < s_{pre}.limit$$

Intuitively, the pre-condition rule holds because the length of stack is the same as the size of the corresponding bag and the limit of the stack is the same as the bound for the bag. Here is an informal proof with slightly more detail:

1. A maps the *items* sequence to the *elems* multiset by putting all elements of the sequence into the multiset. Therefore the length of the sequence $s_{pre}.items$ is equal to the size of the multiset $A(s_{pre}).elems$.
2. Also, A maps the limit of the stack to the bound of the bag so that $s_{pre}.limit = A(s_{pre}).bound$.
3. From *put*'s pre-condition we know $length(s_{pre}.items) < s_{pre}.limit$.
4. *push*'s pre-condition holds by substituting equals for equals.

[5] Note that we are reasoning in terms of the *values* of the object, s, and that b and s refer to the same object.

Notice the role of the abstraction function in this proof. It allows us to relate stack and bag values, and therefore we can relate predicates about bag values to those about stack values and vice versa. Also, note how we depend on A being a function (in step (4) where we use the substitutivity property of equality).

The predicate rule requires that we show *push*'s predicate implies *put*'s:

length $(s_{pre}.items) < s_{pre}.limit \Rightarrow$
$\quad s_{post} = < s_{pre}.items \,||\, [i], s_{pre}.limit > \wedge$ **modifies** s
$\Rightarrow$
$size(A(s_{pre}).elems) < A(s_{pre}).bound \Rightarrow$
$\quad A(s_{post}) = < A(s_{pre}).elems \cup \{i\}, A(s_{pre}).bound > \wedge$ **modifies** s

To show this, we note first that since the two pre-conditions are equivalent, we can ignore them and deal with the post-conditions directly. (Thus we are proving America's stronger post-condition rule in this case.) Next, we deal with the **modifies** and **ensures** parts separately. The **modifies** part holds because the same object is mentioned in both specifications. (Recall that the modifies clause indicates that all objects other than those listed cannot be modified.) The **ensures** part follows directly from the definition of the abstraction function.

Finally, we use the extension mapping to define *swap_top*'s effect. As stated earlier, it has the same effect as that described by the program, π, in which a call to *pop* is followed by one to *push*:

E(s.swap_top(i)) = s.pop(); s.push(i)

Showing the extension rule is just like showing that an implementation of a procedure satisfies the procedure's specification, except that we do not require equal values at the end, but just values that map via A to the same abstract value. (In fact, such a proof is identical to a proof showing that an implementation of an operation of an abstract data type satisfies its specification [13].) In doing the reasoning we rely on the specifications of the methods used in the program. Here is an informal argument for *swap_top*. We note first that since s.swap_top(i) terminates normally, so does the call on s.pop() (their pre-conditions are the same). *Pop* removes the top element, reducing the size of the stack so that *push*'s pre-condition holds, and then *push* puts i on the top of the stack. The result is that the top element has been replaced by i. Thus, $s_{\rho_2} = s_{\psi}$, where ρ_2 is the termination state if we run *swap_top* and ψ is the termination state if we run π. Therefore $A(s_{\rho_2}) = A(s_{\psi})$, since A is a function.

In the arguments given above, we have taken pains to describe the steps of the proof. In fact, most parts of these proofs are obvious and can be done by inspection. The only interesting issues are (1) the definition of the abstraction function, and (2) the definition of the extension map for the new methods that are mutators. The arguments about the methods and extension rules are usually trivial.

4 Type Hierarchies

The constraint we impose on subtypes is very strong and raises a concern that it might rule out many useful subtype relations. To address this concern we applied our method to a number of examples. We found that our technique captures what people want from a hierarchy mechanism (the so-called "is-a" relation in the literature), but we also discovered some surprises.

The examples led us to classify subtype relationships into two broad categories. In the first category, the subtype extends the supertype by providing additional methods and/or additional "state." In the second, the subtype is more constrained than the supertype. We discuss these relationships below.

4.1 Extension Subtypes

A subtype extends its supertype if its objects have extra methods in addition to those of the supertype. Abstraction functions for extension subtypes are onto, i.e., the range of the abstraction function is the set of all legal values of the supertype. The subtype might simply have more methods; in this case the abstraction function is one-to-one. Or its objects might have more "state," i.e., they might record information that is not present in objects of the supertype; in this case the abstraction function is many-to-one.

As an example of the one-to-one case, consider a type intset (for set of integers). Intset objects have methods to *insert* and *delete* elements, to *select* elements, and to provide the *size* of the set. A subtype, intset2, might have more methods, e.g., *union*, *is_empty*. Here there is no extra state, just extra methods. Explanations must be provided for the extra methods using the extension map E, but for all but mutators, these are trivial. Thus, if *union* is a pure constructor, it has the empty explanation, ϵ; otherwise it requires a non-trivial explananation, e.g., in terms of *insert*.

Sometimes it is not possible to find an extension map and therefore there is no subtype relationship between the two types. For example, intset is not a subtype of fat_set, where fat_set objects have only *insert*, *select*, and *size* methods; fat_sets only grow while intsets grow and shrink. Intuitively intset cannot be a subtype of fat_set because it does not preserve various history properties of fat_set. For example, we can prove that once an element is inserted in a fat_set, it remains forever. More formally, for any computation, c:

$$\forall s : fat_set, \rho, \psi : State \;.\; [\rho < \psi \wedge s \in dom(\rho)] \Rightarrow \\ [\forall x : int \;.\; x \in s_\rho \Rightarrow x \in s_\psi]$$

where $\rho < \psi$ means ρ precedes ψ in c. This theorem does not hold for intset. The attempt to construct a subtype relation fails because no extension map can be given to explain the effect of intset's *delete* method.

As a simple example of a many-to-one case, consider immutable pairs and triples. Pairs have methods that fetch the first and second elements; triples have these methods plus an additional one to fetch the third element. Triple is a subtype of pair and so is semi-mutable triple with methods to fetch the first, second,

and third elements and to replace the third element. Here, E(x.replace(e)) = ϵ because the modification is not visible to users of the supertype. This example shows that it is possible to have a mutable subtype of an immutable supertype, provided the mutations are invisible to users of the supertype.

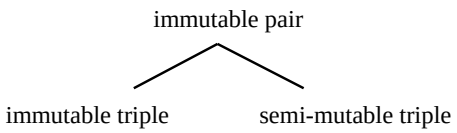

Fig. 3. Pairs and Triples

Mutations of a subtype that would be visible through the methods of an immutable supertype are ruled out. For example, an immutable sequence, which allows its elements to be fetched but not stored, is not a supertype of mutable array, which provides a *store* method in addition to the sequence methods. For sequences we can prove elements do not change; this is not true for arrays. The attempt to construct the subtype relation will fail because there is no way to explain the *store* method via an extension map.

Many examples of subtypes that are extensions are found in the literature. One common example concerns persons, employees, and students. A person object has methods that report its properties such as its name, age, and possibly its relationship to other persons (e.g., its parents or children). Student and employee are subtypes of person; in each case they have additional properties, e.g., a student id number, an employee employer and salary. In addition, type student_employee is a subtype of both student and employee (and also person, since the subtype relation is transitive). In this example, the subtype objects have more state than those of the supertype as well as more methods.

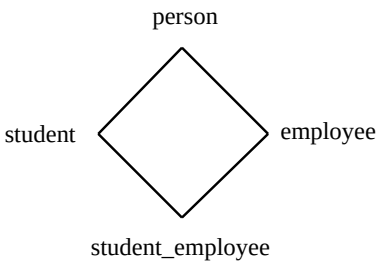

Fig. 4. Person, Student, and Employee

Another example from the database literature concerns different kinds of ships [12]. The supertype is ordinary ships with methods to determine such

things as who is the captain and where the ship is registered. Subtypes contain more specialized ships such as tankers and freighters. There can be quite an elaborate hierarchy (e.g., tankers are a special kind of freighter). Windows are another well-known example [11]; subtypes include bordered windows, colored windows, and scrollable windows.

Common examples of subtype relationships are allowed by our definition provided the *equal* method (and other similar methods) are defined properly in the subtype. Suppose supertype τ provides an *equal* method and consider a particular call x.equal(y). The difficulty arises when x and y actually belong to σ, a subtype of τ. If objects of the subtype have additional state, x and y may differ when considered as subtype objects but ought to be considered equal when considered as supertype objects.

For example, consider immutable triples $x =< 0, 0, 0 >$ and $y =< 0, 0, 1 >$. Suppose the specification of the *equal* method for pairs says:

equal = **proc** (q: pair) **returns** (bool)
ensures Returns true if p.first = q.first and p.second = q.second; false, otherwise.

(We are using p to refer to the method's object.) However, for triples we would expect the following specification:

equal = **proc** (q: triple) **returns** (bool)
ensures Returns true if p.first = q.first, p.second = q.second, and p.third = q.third; false, otherwise.

If a program using triples had just observed that x and y differ in their third element, we would expect x.equal(y) to return "false." However, if the program were using them as pairs, and had just observed that their first and second elements were equal, it would be wrong for the *equal* method to return false.

The way to resolve this dilemma is to have two *equal* methods in triple:

pair_equal = **proc** (p: pair) **returns** (bool)
ensures Returns true if p.first = q.first and p.second = q.second; false, otherwise.

triple_¡equal = **proc** (p: triple) **returns** (bool)
ensures Returns true if p.first = q.first, p.second = q.second, and p.third = q.third; false, otherwise.

One of them (*pair_equal*) simulates the *equal* method for pair; the other (*triple_equal*) is a method just on triples.

The problem is not limited to equality methods. It also affects methods that "expose" the abstract state of objects, e.g., an *unparse* method that returns a string representation of the abstract state of its object. x.unparse() ought to return a representation of a pair if called in a context in which x is considered to be a pair, but it ought to return a representation of a triple in a context in which x is known to be a triple (or some subtype of triple).

The need for several equality methods seems natural for realistic examples. For example, asking whether e1 and e2 are the same person is different from asking if they are the same employee. In the case of a person holding two jobs, the answer might be true for the question about person but false for the question about employee.

4.2 Constrained Subtypes

The second type of subtype relation occurs when the subtype is more constrained than the supertype either in what its methods do or in the values of objects or both. In this case, the supertype specification will always be nondeterministic; its purpose is to allow variations in behavior among its subtypes. Subtypes constrain the supertype by reducing or eliminating the nondeterminism. The abstraction function is usually into rather than onto. The subtype may extend those supertype objects that it simulates by providing additional methods and/or state.

A very simple example concerns elephants. Elephants come in many colors (realistically grey and white, but we will also allow blue ones). However all albino elephants are white and all royal elephants are blue. Figure 5 shows the elephant hierarchy. The set of legal values for regular elephants includes all elephants whose color is grey or blue or white. The set of legal values for royal elephants is a subset of those for regular elephants and hence the abstraction function is into. The situation for albino elephants is similar.

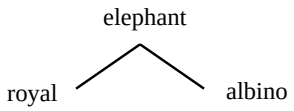

Fig. 5. Elephant Hierarchy

Though the value sets are different, the specifications for the *get_color* method for the supertype and its subtypes might actually be the same:

get_color = **proc** () **returns** (color)
ensures Returns the color of e.

where e is the elephant object. If e is a regular elephant then the caller of *get_color* should expect one of three colors to be returned; if e is a royal elephant, the caller should expect only blue. Alternatively, the nondeterminism in the specification of *get_color* for regular elephants might be made explicit:

get_color = **proc** () **returns** (color)
ensures Returns grey or blue or white.

Then we would need to change the specification for the method for royal elephants to:

get_color = **proc** () **returns** (color)
ensures Returns blue.

Notice that it would be wrong for the post-condition of *get_color* for regular elephants to say just "Returns grey" because then the predicate rule would not hold when showing that royal elephant is a subtype of elephant.

Not only must the specifications of corresponding methods relate appropriately, but any invariant property that holds for supertype objects must hold for subtype objects. Suppose we removed the nondeterminism in the specification for regular elephants, defining the value set for regular elephants to be just those elephants whose color is grey. Then the theorem stating all elephants are grey:

$$\forall e : elephant\ \forall \rho : State\ .\ [e \in dom(\rho) \Rightarrow e_\rho.color = grey]$$

would hold of all regular elephants but not for any of its subtype objects. Instead, our weaker theorem does hold for regular elephants and its subtypes:

$$\forall e : elephant\ \forall \rho : State\ .\ [e \in dom(\rho) \Rightarrow e_\rho.color = grey \vee e_\rho.color = blue \vee e_\rho.color = white]$$

This simple example has led others to define a subtyping relation that requires non-monotonic reasoning [18], but we believe it is better to use a nondeterministic specification and straightforward reasoning methods. However, the example shows that a specifier of a type family has to anticipate subtypes and capture the variation among them in a nondeterministic specification of the supertype.

Another similar example concerns geometric figures. At the top of the hierarchy is the polygon type; it allows an arbitrary number of sides and angles of arbitrary sizes. Subtypes place various restrictions on these quantities. A portion of the hierarchy is shown in Figure 6.

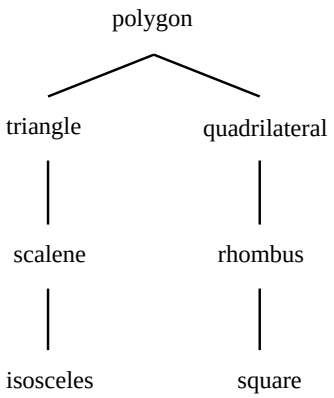

Fig. 6. Polygon Hierarchy

The bag type informally discussed in Section 3.1 is nondeterministic in two ways. As discussed earlier, the specification of *get* is nondeterministic because it

does not constrain which element of the bag is removed. This nondeterminism allows stack to be a subtype of bag: The specification of *pop* constrains the nondeterminism. We could also define a queue that is a subtype of bag; its *dequeue* method would also constrain the nondeterminism of *get* but in a different way than *pop* does.

In addition, since the actual value of the bound for bags was not constrained, it can be any natural number, thus allowing subtypes to have different bounds. This nondeterminism shows up in the specification of *put*, where we do not say what specific bound value causes the call to fail. Therefore, a user of *put* must be prepared for a failure unless it is possible to deduce from past evidence, using the history property that the bound of a bag does not change, that the call will succeed. A subtype of bag might constrain the bound to a fixed value, or to a smaller range. Several subtypes of bag are shown in Figure 7; largebags are essentially unbounded bags since their bound (fixed at creation) is ∞, and mediumbags have various bounds, so that this type might have its own subtypes, e.g., bag_150, containing all bags with bound equal to 150.

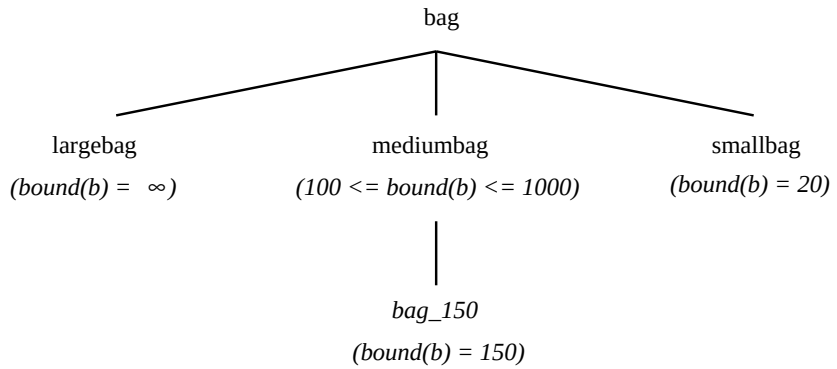

Fig. 7. A Type Family for Bags

The bag hierarchy may seem counterintuitive, since we might expect that bags with smaller bounds should be subtypes of bags with larger bounds. For example, we might expect bag_150 to be a subtype of largebag. However, the specifications for the two types are incompatible. For largebags we can prove that the bound of every bag is ∞, which is clearly not true for bag_150. Furthermore, this difference is observable via the methods: It is legal to call the *put* method on a largebag whose size is greater than or equal to 150, but the call is not legal for a bag_150. Therefore the pre-condition rule is not satisfied.

Although the bag type can have subtypes with different constraints on the bounds, it is not a valid supertype of a dynamic_bag type where the bounds of the bags can change dynamically. Dynamic_bags would have an additional method, *change_bound*, for object b:

change_bound = **proc** (n: int)
requires n is greater than or equal to the size of b.
modifies b
ensures Sets the bound of b to n.

Change_bound is a mutator for which no explanation via an extension map is possible. Note that we can prove that the bound of a bag object does not vary; clearly this is not true for a dynamic_bag object.

If we wanted a type family that included both dynamic_bag and bag, we would need to define a supertype in which the bound is allowed, but not required, to vary. Figure 8 shows the new type hierarchy where the *change_bound* method for varying_bag looks like:

change_bound = **proc** (n: int)
requires n is greater than or equal to the size of b.
modifies b
ensures Either sets b's bound to n or keeps it the same.

Not only is this specification nondeterministic about the bounds of bag objects, but the specification of the *change_bound* method is nondeterministic: The method may change the bound to the new value, or it may not. This nondeterminism is resolved in its subtypes; bag (and its subtypes) provide a *change_bound* method that leaves the bound as it was, while dynamic_bag changes it to the new bound. Note that for bag to be a subtype of varying_bag, it must have a *change_bound* method (in addition to its other methods).

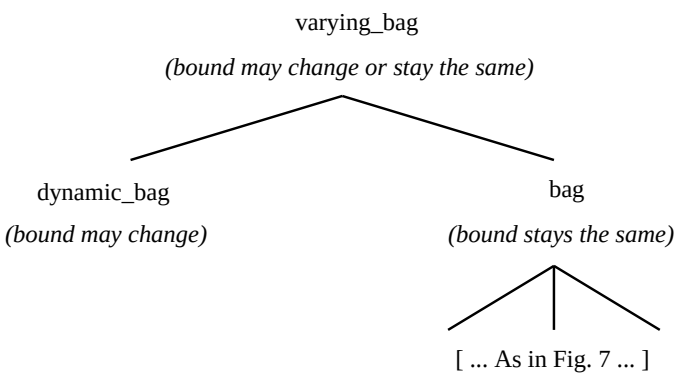

Fig. 8. Another Type Family for Bags

In the case of the bag family illustrated in Figure 7, all types in the hierarchy might actually be implemented. However, sometimes the supertypes are not intended to be implemented. These *virtual types* serve as placeholders for specific subtypes that are intended to be implemented; they let us define the properties all the subtypes have in common. Varying_bag is an example of such a type.

Virtual types are also needed when we construct a hierarchy for integers. Smaller integers cannot be a subtype of larger integers because of observable differences in behavior; for example, an overflow exception that would occur when adding two 32-bit integers would not occur if they were 64-bit integers. However, we clearly would like integers of different sizes to be related. This is accomplished by designing a nondeterministic, virtual supertype that includes them. Such a hierarchy is shown in Figure 9, where integer is a virtual type. Here integer types with different sizes are subtypes of integer. In addition, small integer types are subtypes of regular_int, another virtual type. Such a hierarchy might have a structure like this, or it might be flatter by having all integer types be direct subtypes of integer.

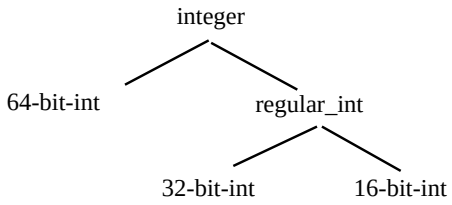

Fig. 9. Integer Family

5 Related Work

Research on defining subtype relations can be divided into two categories: work on the "syntactic" notion of subtyping and work on the "semantic" notion. We clearly differ from the syntactic notion, formally captured by Cardelli's contra/covariance rules [5] and used in languages like Trellis/Owl [24], Eiffel [21], POOL [2], and to a limited extent Modula-3 [23]. Our rules place constraints not just on the signatures of an object's methods, but also on their semantic behavior as described in type specifications. Cardelli's rules are a strict subset of ours (ignoring higher-order functions).

Our semantic notion differs from the others for two main reasons: We deal with mutable abstract types and we allow subtypes to have additional methods. We discuss this related work in more detail below. We also mention how our work is related to models for concurrent processes.

Our work is most closely related to that of America [3] who uses the stronger pre- and post-condition rules as discussed in Section 3. (Meyer also uses these rules for Eiffel [21], although here the pre- and post-conditions are given "operationally," by providing a program to check them, rather than assertionally.) It is also related to work by Cusack [7] who defines subtyping in terms of strengthening state invariants. However, neither author considers the problems introduced by extra mutators. Thus, by not considering history properties at all, they allow

certain subtype relations that we forbid (e.g., set could be a subtype of fat_set in these approaches).

Our work is also similar to America's in its approach: We expect programmers to reason directly in terms of specifications; we call this approach "proof-theoretic." Most other approaches are "model-theoretic"; programmers are expected to reason in terms of mathematical structures like algebras or categories. We believe a proof-theoretic approach is better because it is much more accessible to programmers.

The emphasis on semantics of abstract types is a prominent feature of the work by Leavens. We go further by addressing mutable abstract types. In his Ph.D. thesis [15] Leavens defines a model-theoretic semantic notion of subtyping. He defines types in terms of algebras and subtyping in terms of a *simulation relation* between them. Further work by Leavens and Weihl showing how to verify programs with subtypes uses Hoare-style reasoning as we do [16]. Again, their work is restricted to immutable types. Their *simulation relations* map supertype values down to subtype values; hence, they do reasoning in the subtype value space. In contrast we use our abstraction function to map values up to the supertype value space. We can rely on the substitutivity property of equality; they cannot. Indeed, in our proofs we depend on A being a function.

Bruce and Wegner give a model-theoretic semantic definition of subtyping (in terms of algebras) but also do not deal with mutable types [4]. Like Leavens they model types in terms of algebras; like us they define *coercion functions* with the substitution property. They cannot handle mutable types and are not concerned with reasoning about programs directly.

In his 1992 Master's thesis [9], Dhara extends Leavens' thesis work to deal with mutable types. Again, his approach is model-theoretic and based on simulation relations; moreover, because of a restriction on aliasing in his model, his definition disallows certain subtype relations from holding that we could allow. Dhara has no counterpart to our extension rule for extra mutators, and no techniques for proving subtype relations.

To our knowledge, Utting is the only other researcher to take a proof-theoretic approach to subtyping [25]. His formalism is cast in the refinement calculus language [22], an extension of Dijkstra's guarded command language [10]. Utting makes a big simplifying assumption: he does not allow data refinement between supertype and subtype value spaces. Our use of abstraction functions directly addresses this issue, which intuitively is the heart of any subtyping relation.

Finally, our extension rule is related to the more general work done on relating the behaviors of two different concurrent processes. Cusack has investigated the meaning of subtyping for CSP processes [6] and (with Rudkin and Smith) LOTOS processes [8], but takes a model-theoretic approach. Abadi and Lamport's *refinement mappings* [1] are akin to our extension mappings, but have not yet been applied in the context of subtyping.

6 Conclusions

In this paper we have defined a method for reasoning about whether one type is a subtype of another based on the semantic properties of the two types. An object's type determines both a set of legal values and an interface with its environment (through calls on its methods). Thus, we are interested in preserving properties about supertype values and methods when designing a subtype. We are particularly interested in an object's observable behavior (state changes), thus motivating our focus on mutable types and mutators. Ours is the first approach to provide a way of determining the acceptability of the "extra" mutators.

Our approach guarantees that the subtype preserves all the invariant and history properties of the supertype. This very strong definition ensures that if one user reasons about a shared object using properties that hold for its apparent type, that reasoning will be valid even if the object actually belongs to a subtype and is manipulated by other users using the subtype methods. Our definition of subtyping works even in a very general environment in which possibly concurrent users share mutable objects.

The constraint we impose on subtypes is very strong and raises a concern that it might rule out many useful subtype relations. To address this concern we applied our method to a number of examples. We found that it is not difficult to define useful type hierarchies that fit our constraint, and identified two kinds of type families, those in which the subtypes extend the supertype by adding extra state and methods, and those in which the subtypes constrain the supertype by removing or eliminating nondeterminism.

In developing our definition, we were motivated primarily by pragmatics. Our intention is to capture the intuition programmers apply when designing type hierarchies in object-oriented languages. However, intuition in the absence of precision can often go astray or lead to confusion. This is why it has been unclear how to organize certain type hierarchies such as integers. Our definition sheds light on such hierarchies and helps in uncovering new designs. It also supports the kind of reasoning needed to ensure that programs that work correctly using the supertype continue to work correctly with the subtype.

We believe that programmers will find our definition relatively easy to apply and expect it to be used primarily in an informal way. The essence of a subtype relationship is expressed in the mappings. We hope that the mappings will be defined as part of giving type and subtype specifications, in much the same way that abstraction functions and representation invariants are given as comments in a program that implements an abstract type. The proofs are usually trivial and can be done by inspection. Such a property is necessary if we expect formal definitions to be applied in daily use.

Acknowledgments

We thank Gary Leavens for a helpful discussion of subtyping and pointers to related work. In addition, Gary, Mark Day, Sanjay Ghemawat, John Guttag,

Deborah Hwang, Greg Morrisett, Eliot Moss, John Reynolds, Bill Weihl, Amy Moormann Zaremski, and the referees gave useful comments on earlier versions of this paper.

This research was supported for Liskov in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136 and in part by the National Science Foundation under Grant CCR-8822158; for Wing, by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Digital Equipment Corp. Sys. Research Ctr. Tech Rpt. 29, 1988.
2. America, P.: A parallel object-oriented language with inheritance and subtyping. ACM SIGPLAN **25** (1990) 161-168.
3. America, P.: Designing an object-oriented programming language with behavioural subtyping. Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, Springer-Verlag Lec. Notes in Com. Sci. **489** (1991) 60-90.
4. Bruce, K. B., Wegner, P.: An algebraic model of subtypes in object-oriented languages (draft). ACM SIGPLAN Notices **21** (1986).
5. Cardelli, L.: A semantics of multiple inheritance. Info. and Computation **76** (1988) 138-164.
6. Elspeth Cusack: Refinement, conformance, and inheritance. Formal Aspects of Computing. 3(2), (1991), 129-141.
7. Elspeth Cusack: Inheritance in object oriented Z. Proceedings of ECOOP '91, 1991.
8. Elspeth Cusack, Steve Rudkin, and Chris Smith: An object oriented interpretation of LOTOS. Formal Description Techniques, II, S.T. Vuong (ed.). Elsevier Science Publishers B.V. (North-Holland) (1990), 211-226.
9. Dhara, K. K.: Subtyping among mutable types in object-oriented programming languages. Iowa State University Ames, Iowa, 1992.
10. Dijkstra, E. W.: A Discipline of Programming. Prentice Hall, New York, 1976.
11. Halbert, D. C., O'Brien, P. D.: Using types and inheritance in object-oriented programming. IEEE Software (1987) 71-79.
12. Hammer, M., McLeod, D.: A semantic database model. ACM Trans. Database Systems **6** (1981) 351-386.
13. Hoare, C. A. R.: Proof of correctness of data representations. Acta Informatica **1** (1972) 271-281.
14. Kapur, K.: Towards a theory of abstract data types. Tech. Rpt. 237, MIT Lab. for Computer Science, Cambridge, MA, 1980.
15. Leavens, G.: Verifying object-oriented prograsm that use subtypes. Tech. Rpt. 439, MIT Lab. for Computer Science, Cambridge, MA, 1989.
16. Leavens, G., Weihl, W. E.: Reasoning about object-oriented programs that use subtypes. ECOOP/OOPSLA '90 Proceedings, 1990.
17. Leavens, G., Weihl, W.E.: Subtyping, modular specification, and modular verification for applicative object-oriented programs. (forthcoming).

18. Lipeck, U.: Semantics and usage of defaults in specifications. Foundations of Information Systems Specification and Design, Hans-Dieter Ehrich and Joseph A Goguen and Amilcar Sernadas Dagstuhl Seminar 9212, Rpt. 35, 1992.
19. Liskov, B., Guttag, J.: Abstraction and Specification in Program Design. McGraw Hill and MIT Press, 1985.
20. Liskov, B., Wing, J. M.: Family Values: A Semantic Notion of Subtyping. MIT-LCS-TR-562, Tech. Rpt., December 1992. Also published as CMU-CS-92-220 TR.
21. Meyer, B.: Object-oriented Software Construction. Prentice Hall, New York, 1988.
22. Morgan, C.: Programming from Specifications. Prentice Hall, New York, 1990.
23. Nelson, G. (Ed.): Systems Programming with Modula-3. Prentice Hall, New York, 1991.
24. Schaffert, C., Cooper, T., Wilpolt, C.: Trellis: object-based environment language reference manual. Dig. Equip. Corp. Eastern Research Lab., Tech Rpt. 372, 1985.
25. Utting, M.: An object-oriented refinement calculus with modular reasoning. University of New South Wales, Australia, 1992.

This article was processed using the LaTeX macro package with LLNCS style