MIT/LCS/TR-562.b
(replaces TR-562)

# FAMILY VALUES: A BEHAVIORAL NOTION OF SUBTYPING

August 1993

Barbara Liskov
(MIT, Laboratory for Computer Science)
Jeannette M. Wing
(Carnegie Mellon University, School of Computer Science)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE
CAMBRIDGE, MA 02139

MIT/LCS/TR-562-1

# FAMILY VALUES: A BEHAVIORAL NOTION OF SUBTYPING

Barbara Liskov
Jeannette M. Wing[1]

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, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

AUGUST 1993

[1]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

# Family Values: A Behavioral Notion of Subtyping

Barbara Liskov[*]
MIT Laboratory for Computer Science
Cambridge, MA 02139

Jeannette M. Wing[†]
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

**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 takes the position that the relationship should ensure that any property proved about supertype objects also holds for its subtype objects. It presents two ways of defining the subtype relation, each of which meets this criterion, and each of which is easy for programmers to use. The subtype relation is based on the specifications of the sub- and supertypes; the paper presents a way of specifying types that makes it convenient to define the subtype relation. The paper also discusses the ramifications of this notion of subtyping on the design of type families.

Keywords: subtype, object-oriented design, abstraction function, extensible types, mutable types, formal specifications, semantics, Larch.

[*]Supported 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

[†]Supported in part 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.

## 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 behavior of the objects of the two types: the objects of the subtype ought to behave the same as those of the supertype as far as anyone or any program using supertype objects can tell.

For example, in strongly typed object-oriented languages such as Simula 67[9], C++[35], Modula-3[32], and Trellis/Owl[33], subtypes are used to broaden the assignment statement. An assignment

x: T := E

is 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 of the contra/covariance rules[3, 33, 5]. However, these 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, stacks and queues 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: properties that can be proved using the specification of an object's presumed type should hold even though the object is actually a member of a subtype of that type. This paper's main contribution is to provide two general, yet easy to use, definitions of the subtype relation that precisely capture this subtype requirement. Our definitions extend earlier work, including the most closely related work done by America[2], by allowing subtypes to have more methods than their supertypes. They apply even in a very general environment in which possibly concurrent users share mutable objects. Our approach is also constructive: One can prove whether a subtype relation holds by proving a small number of simple lemmas based on the specifications of the two types.

Our paper makes two other contributions. First, it provides a way of specifying object types that allows a type to have multiple implementations and makes it convenient to define the subtyping relation. Our specifications are formal, which means that they have a precise mathematical meaning that serves as a firm foundation for reasoning. Our specifications can also be used informally as described in [27].

Second, it explores 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). However, type families can be defined that group such related types together.

The paper is organized as follows. Section 2 discusses in more detail what we require of our subtype relation and provides the motivation for our approach. Next we describe our model of computation and then present our specification method. Section 5 presents our two definitions of subtyping and Section 6 discusses the ramifications of our approach on designing type hierarchies. We describe related work in Section 7 and then close with a summary of contributions.

## 2 Motivation

To motivate the basic idea behind our notion of subtyping, let's look at an example. Consider a bounded bag type that provides a *put* method that inserts elements into a bag and a *get* method that removes an arbitrary element from 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.

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.

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 [19]. 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 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*? Most other definitions of the subtype relation have ignored such "extra" methods, and it is perfectly adequate do so when procedures are considered in isolation and there is no aliasing. In such a constrained situation, a program that uses an object that is apparently a bag but is actually a stack will never call the extra methods, and therefore their behavior is irrelevant. However, we cannot ignore extra methods in the presence of aliasing, and also in a general computational environment that allows sharing of mutable objects by multiple users.

Consider first the case of aliasing. The problem here is that within a procedure an object is accessible by more than one name, so that modifications using one of the names are visible when the object is accessed using the other name. For example, suppose $\sigma$ is a subtype of $\tau$ and that variables

x: $\tau$
y: $\sigma$

both denote the same object (which must, of course, belong to $\sigma$ or one of its subtypes). When the object is accessed through x, only $\tau$ methods can be called. However, when it is used through y, $\sigma$ methods can be called and the effects of these methods are visible later when the object is accessed via x. To reason about the use of variable x using the specification of its type $\tau$, we need to impose additional constraints on the subtype relation.

Now consider the case of an environment of shared mutable objects, such as is provided by object-oriented databases (e.g., Thor [26] and Gemstone [29]). (In fact, it was our interest in Thor that motivated us to study the meaning of the subtype relation in the first place.) In such systems, there is a universe containing shared, mutable objects and a way of naming those objects. In general, lifetimes of objects may be longer than the programs that create and access them (i.e., objects might be persistent) and users (or programs) may access objects concurrently and/or aperiodically for varying lengths of time. Of course there is a need for some form of concurrency control in such an environment. We assume such a mechanism is in place, and consider a computation to be made up out of atomic units (i.e., transactions) that exclude one another. The transactions of different computations can be interleaved and thus one computation is able to observe the modifications made by another.

If there were subtyping in such an environment the following situation might occur. A user installs a directory object that maps string names to bags. Later, a second user enters a stack into the directory under some string name; such binding is analogous to assigning a subtype object to a variable of the supertype. After this, both users occasionally access the stack object. The second user knows it is a stack and accesses it using stack methods. The question is: What does the first user need to know in order for his or her programs to make sense?

We think it ought to be sufficient for a user to only know about the "apparent type" of the object; the subtype ought to preserve any properties that can be proved about the supertype. We are concerned only with *safety* properties ("nothing bad happens"). 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, an invariant property of a bag is that its size is always less than its bound; a history property is that its bound does not change. We might also want to prove *liveness* properties ("something good eventually happens"), e.g., the size of a bag will eventually reach the bound, but our focus here will be just on safety properties.

Thus the first user ought to be able to reason about his or her use of the stack object using invariant and history properties of bag. Both of our definitions of subtype assume a type specification includes an explicit **invariant** clause that states the type invariants that must be preserved by any of it subtypes. Our two definitions differ in the way they handle extra methods, and thus in their way of ensuring that history properties are preserved:

- Our first definition deals with the history properties directly. We add to a type's specification a **constraint** clause that captures exactly those history properties of a type that must be preserved by any of its subtypes, and we prove that each of the type's methods preserves the constraint. Showing that $\sigma$ is a subtype of $\tau$ requires showing that $\sigma$'s constraint implies $\tau$'s (under the abstraction function).

- Our second definition deals with history properties indirectly. For each extra method, we require that an "explanation" be given of how its behavior could be effected by just those methods already defined for the supertype. The explanation guarantees that the extra method does not introduce any behavior that was not already present, and therefore it does not interfere with any history property.

For example, using the first approach we would state constraints for both bags and stacks. In this particular example, the two constraints are identical; both state that the bound of the bag (or stack) does not change. The extra method *swap_top* is permitted because it does not change the stack's bound. Showing that the constraint for stack implies that of bag is trivial. Using the second approach, we would provide an explanation for *swap_top* in terms of existing methods:

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

and we would prove that the explanation program really does simulate *swap_top's* behavior.

In Section 5 we present and discuss these two alternative definitions. First, however, we define our model of computation, and then discuss specifications, since these define the objects, values, and methods that will be related by the subtype relation.

## 3 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 *values* for an object and a set of *methods* that provide the only means to manipulate that object.

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 mappings, an *environment* and a *store*. An environment maps program variables to objects; a store maps objects to values.

$$\begin{aligned} &State = Env \times Store \\ &Env = Var \rightarrow Obj \\ &Store = Obj \rightarrow Val \end{aligned}$$

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

We model a type as a triple, $< O, V, M >$, where $O \subseteq Obj$ is a set of objects, $V \subseteq Val$

is a set of 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 $\tau$ return new objects of type $\tau$; observers return results of other types; mutators modify the values of objects of type $\tau$. A type is *mutable* if any of its methods is a mutator. 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 the sequence 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 a sequence of alternating states and statements starting in some initial state, $\rho_0$:

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

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[1]: 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:

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

## 4 Specifications

### 4.1 Type Specifications

A type specification includes the following information:

- The type's name;
- A description of the type's value space;
- For each of the type's methods:

[1]This model is based on CLU semantics[13].

– Its name;
– Its signature (including signaled exceptions);
– Its behavior in terms of pre-conditions and post-conditions.

Note that the creators are missing. Creators are specified separately to make it easy for a type to have multiple implementations, to allow subtypes to have different creators from their supertypes, and to make it more convenient to define subtypes. We show how to specify creators in Section 4.2. However, the absence of creators means that data type induction cannot be used to reason about invariant properties. In Section 4.3 we discuss how we make up for this loss by adding invariants to type specifications.

In our work we use formal specifications in the two-tiered style of Larch [16]. The first tier defines *sorts*, which are used to define the value spaces of objects. In the second tier, Larch *interfaces* are used to define types. For example, Figure 1 gives a specification for a bag type whose objects have methods *put*, *get*, *card*, and *equal*. The **uses** clause defines the value space for the type by identifying a sort. The clause in the figure indicates that values of objects of type bag are denotable by terms of sort $B$ introduced in the BBag specification; a value of this sort is a pair, $< elems, bound >$, where *elems* is a mathematical multiset of integers and *bound* is a natural number. The notation $\{\ \}$ stands for the empty multiset, $\cup$ is a commutative operation on multisets that does not discard duplicates, and $\mid x \mid$ is a cardinality operation that returns the total number of elements in the multiset $x$.

The body of a type specification provides a specification for each method. Since a method's specification needs to refer to the method's object, we introduce a name for that object in the **for all** line. A **requires** clause gives a method's pre-condition; e.g., *put*'s pre-condition checks to see that adding an element will not grow the bag beyond its bound. If the clause is missing, the pre-condition is trivially "true." A **modifies** $x_1, \ldots, x_n$ clause is shorthand for the predicate:

$$\forall\ x \in (dom(pre) - \{x_1, \ldots, x_n\})\ .\ x_{pre} = x_{post}$$

which says only objects listed may change in value. A **modifies** clause is a strong statement about all objects not explicitly listed, i.e., their values may not change. If there is no **modifies** clause then nothing may change. The post-condition is the conjunction of the **modifies** and **ensures** clauses; e.g., *put*'s post-condition says that the bag's value changes by the addition of its integer argument. For method $m$, we write *m.pre* to denote its pre-condition and *m.post* its post-condition.

```
bag = **type**

  uses BBag (bag for $B$)
  for all $b$: bag

        $put$ = proc ($i$: int)
                requires | $b_{pre}.elems$ | < $b_{pre}.bound$
                modifies $b$
                ensures  $b_{post}.elems = b_{pre} \cup \{i\} \;\wedge\; b_{post}.bound = b_{pre}.bound$

        $get$ = proc ( ) returns (int)
                requires $b_{pre} \neq \{\}$
                modifies $b$
                ensures  $b_{post}.elems = b_{pre}.elems - \{result\} \wedge result \in b_{pre}.elems \wedge$
                         $b_{post}.bound = b_{pre}.bound$

        $card$ = proc ( ) returns (int)
                ensures  $result = | b_{pre}.elems |$

        $equal$ = proc ($a$: bag) returns (bool)
                ensures  $result = (a = b)$

  end bag
```

Figure 1: A Type Specification for Bags

In the **requires** and **ensures** clauses $x$ stands for an object, $x_{pre}$ for its value in the initial state, and $x_{post}$ for its value in the final state.[2] Distinguishing between initial and final values is necessary only for mutable types, so we suppress the subscripts for parameters of immutable types (like integers). We need to distinguish between an object, $x$, and its value, $x_{pre}$ or $x_{post}$, because we sometimes need to refer to the object itself, e.g., in the *equal* method, which determines whether two (mutable) bags are identical. *Result* is a way to name a method's result parameter.

Methods may terminate normally or exceptionally; the exceptions are listed in a **signals** clause in the method's header. For example, an alternative specification for the *get* method is

```
    $get$ = proc ( ) returns (int) signals (empty)
          modifies $b$
          ensures if $b_{pre}.elems = \{\ \}$ then signal empty
                  else $b_{post}.elems = b_{pre}.elems - \{result\} \wedge result \in b_{pre}.elems \wedge$
                  $b_{post}.bound = b_{pre}.bound$
```

[2]Referring to an object's final value is meaningless in pre-conditions, of course.

*create*= **proc** ($n$: int) **returns** (bag)
  **requires** $n \geq 0$
  **ensures** **new**$(result) \wedge result_{post} = < \{\}, n >$

*create_small* = **proc** ( ) **returns** (bag)
  **ensures** **new**$(result) \wedge result_{post} = < \{\}, 100 >$

*create_single* = **proc** ($i$: int) **returns** (bag)
  **ensures** **new**$(result) \wedge result_{post} = < \{i\}, 100 >$

Figure 2: Creator Specifications for Bags

## 4.2 Specifying Creators

Objects are created and initialized through creators. Figure 2 shows specifications for three different creators for bags. The first creator creates a new empty bag whose bound is its integer argument. The second and third creators fix the bag's bound to be 100. The third creator uses its integer argument to create a singleton bag. The assertion **new**($x$) stands for the predicate:

$$x \in dom(post) - dom(pre)$$

Recall that objects are never destroyed so that $dom(pre) \subseteq dom(post)$.

## 4.3 Type Specifications Need Explicit Invariants

By not including creators in type specifications we lose a powerful reasoning tool: data type induction. Data type induction is used to prove type invariants. The base case of the rule requires that each creator of the type establish the invariant; the inductive case requires that each method preserve the invariant. Without the creators, we have no base case, and therefore we cannot prove type invariants!

To compensate for the lack of data type induction, we state the invariant explicitly in the type specification by means of an **invariant** clause; if the invariant is trivial (i.e., identical to "true"), the clause can be omitted. The invariant defines the *legal* values of its type $\tau$. For example, we add

**invariant** $\mid b_\rho.elems \mid \leq b_\rho.bound$

to the type specification of Figure 1 to state that the size of a bounded bag never exceeds its bound. The predicate $\phi(x_\rho)$ appearing in an **invariant** clause for type $\tau$ stands for the predicate:

$$\forall x : \tau,\ \rho : State\ .\ \phi(x_\rho)$$

Any additional invariant properties must follow from the conjunction of the type's invariant and invariants that hold for the entire value space. For example, we could show that the size of a bag is nonnegative because this is true for all mathematical multiset values. Since additional invariants cannot be proved using data type induction, the specifier must be careful to define an invariant that is strong enough to support all desired invariants.

We must show that the specification preserves the invariant. All creators for a type $\tau$ must establish $\tau$'s invariant, $I_\tau$:

- For each creator for type $\tau$, show $I_\tau(result_{post})$.

In addition, each method of the type must *preserve the invariant*. To prove this, we assume each method is called on an object of type $\tau$ with a legal value (one that satisfies the invariant), and show that any value of a $\tau$ object it produces or modifies is legal:

- For each method $m$ of $\tau$, assume $I_\tau(x_{pre})$ and show $I_\tau(x_{post})$.

For example, we would need to show *put*, *get*, *card*, and *equal* each preserves the invariant for bag. Informally the invariant holds because *put*'s pre-condition checks that there is enough room in the bag for another element; *get* either decreases the size of the bag or leaves it the same; *card* and *equal* do not change the bag at all. The proof ensures that methods deal with only legal values of an object's type.

## 5 The Meaning of Subtype

### 5.1 Specifying Subtypes

To state that a type is a subtype of some other type, we simply append a **subtype** clause to its specification. We allow multiple supertypes; there would be a separate **subtype** clause for each. An example is given in Figure 3.

A subtype's value space may be different from its supertype's. For example, in the figure the sort, S, for bounded stack values is defined in BStack as a pair, $< items, limit >$, where *items* is a sequence of integers and *limit* is a natural number. The invariant indicates that the length of the stack's sequence component is less than or equal to its limit. Under the **subtype** clause we define an *abstraction function*, $A$, that relates stack values to bag values by relying on the helping function, $mk\_elems$, that maps sequences to multisets in the obvious manner. (We will revisit this abstraction function in Section 5.2.2.) The **subtype** clause also lets specifiers rename methods of the supertype, e.g., *push* for *put*; all other methods of the supertype are

```
stack = **type**

  **uses** BStack (stack **for** $S$)
  **for all** $s$: stack

        **invariant** $length(s_\rho.items) \leq s_\rho.limit$

        $push$ = **proc** ($i$: int)
                **requires** $length(s_{pre}.items) < s_{pre}.limit$
                **modifies** $s$
                **ensures** $s_{post}.items = s_{pre}.items \mid\mid [\ i\ ] \ \wedge \ s_{post}.limit = s_{pre}.limit$

        $pop$ = **proc** () **returns** (int)
                **requires** $s_{pre}.items \neq [\ ]$
                **modifies** $s$
                **ensures** $result = last(s_{pre}.items) \wedge s_{post}.items = allButLast(s_{pre}.items) \wedge$
                        $s_{post}.limit = s_{pre}.limit$

        $swap\_top$ = **proc** ($i$: int)
                **requires** $s_{pre}.items \neq [\ ]$
                **modifies** $s$
                **ensures** $s_{post}.items = allButLast(s_{pre}.items) \mid\mid [\ i\ ] \ \wedge \ s_{post}.limit = s_{pre}.limit$

        $height$ = **proc** ( ) **returns** (int)
                **ensures** $result = length(s_{pre}.items)$

        $equal$ = **proc** ($t$: stack) **returns** (bool)
                **ensures** $result = (s = t)$

     **subtype of** bag ($push$ **for** $put$, $pop$ **for** $get$, $height$ **for** $card$)
                $\forall st : S \ . \ A(st) = < mk\_elems(st.items), st.limit >$
                    **where** $mk\_elems : Seq \rightarrow M$
                        $\forall i : Int, sq : Seq$
                            $mk\_elems([\ ]) = \{\ \}$
                            $mk\_elems(sq \mid\mid [\ i\ ]) = mk\_elems(sq) \cup \{i\}$

  **end** stack
```

Figure 3: Stack Type

"inherited" without renaming, e.g., *equal.* In the pre- and post-conditions, [ ] stands for the empty sequence, || is concatenation, *last* picks off the last element of a sequence, and *allButLast* returns a new sequence with all but the last element of its argument.

## 5.2 First Definition: Constraint Rule

Our first definition of the subtype relation relies on the addition of some information to specifications, namely a **constraint** clause that states the history properties of the type explicitly[3]; if the constraint is trivial (identically equal to "true"), the clause can be omitted. For example, we add

**constraint** $b_\rho.bound = b_\psi.bound$

to the specification of bag to declare that a bag's bound never changes. We would add a similar clause to stack's specification. As another example, consider a fat_set object that has an *insert* but no *delete* method; fat_sets only grow in size. The constraint for fat_set would be:

**constraint** $\forall\ x : int\ .\ x \in s_\rho \Rightarrow x \in s_\psi$

We can formulate history properties as predicates over state pairs. The predicate appearing in a **constraint** clause is an abbreviation for a history property. For example, bag's constraint expands to the following: For any computation, $c$,

$$\forall b : bag,\ \rho, \psi : State\ .\ [\rho < \psi \wedge b \in dom(\rho)] \Rightarrow [b_\rho.bound = b_\psi.bound]$$

where $\rho < \psi$ means that state $\rho$ precedes state $\psi$ in $c$. Note that we implicitly quantify over all computations, $c$, and do not require that $\psi$ be the immediate successor of $\rho$.

Just as we had to prove that methods preserved the invariant, we must show that they *satisfy the constraint* by proving it for each mutator. We do this by using the history rule:

- *History Rule*: For each mutator $m$ of $\tau$, show $(m.pre\ \wedge\ m.post) \Rightarrow \phi[x_{pre}/x_\rho, x_{post}/x_\psi]$

where $\phi$ is a history property on objects of type $\tau$. $P[a/b]$ stands for predicate $P$ with every occurrence of $b$ replaced by $a$. The constraint replaces the history rule as far as users are concerned: users can make deductions based on the constraint but they cannot reason using the history rule directly.

The formal definition of the subtype relation, $<$, is given in Figure 4. It relates two types, $\sigma$ and $\tau$, each of whose specifications respectively preserves its invariant, $I_\sigma$ and $I_\tau$, and satisfies its constraint, $C_\sigma$ and $C_\tau$. In the methods and constraint rules, since $x$ is an object of type $\sigma$, its value ($x_{pre}$ or $x_{post}$) is a member of $S$ and therefore cannot be used directly in the predicates

[3]The use of the term "constraint" is borrowed from the Ina Jo specification language [34], which also includes constraints in specifications.

DEFINITION OF THE SUBTYPE RELATION, $<:$ $\sigma = < O_\sigma, S, M >$ is a *subtype* of $\tau = < O_\tau, T, N >$ if there exists an abstraction function, $A : S \rightarrow T$, and a renaming map, $R : M \rightarrow N$, such that:

1. The abstraction function respects invariants:
   - *Invariant Rule.* $\forall s : S \, . \, I_\sigma(s) \Rightarrow I_\tau(A(s))$

   A may be partial, need not be onto, but can be many-to-one.

2. Subtype methods preserve the supertype methods' behavior. If $m_\tau$ of $\tau$ is the corresponding renamed method $m_\sigma$ of $\sigma$, the following rules must hold:
   - *Signature rule.*
     - *Contravariance of arguments.* $m_\tau$ and $m_\sigma$ have the same number of arguments. If the list of argument types of $m_\tau$ is $\alpha_i$ and that of $m_\sigma$ is $\beta_i$, then $\forall i \, . \, \alpha_i < \beta_i$.
     - *Covariance of result.* Either both $m_\tau$ and $m_\sigma$ have a result or neither has. If there is a result, let $m_\tau$'s result type be $\gamma$ and $m_\sigma$'s be $\delta$. Then $\delta < \gamma$.
     - *Exception rule.* The exceptions signaled by $m_\sigma$ are contained in the set of exceptions signaled by $m_\tau$.
   - *Methods rule.* For all $x : \sigma$:
     - *Pre-condition rule.* $m_\tau.pre[A(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre$.
     - *Post-condition rule.* $m_\sigma.post \Rightarrow m_\tau.post[A(x_{pre})/x_{pre}, A(x_{post})/x_{post}]$

3. Subtype constraints ensure supertype constraints.
   - *Constraint Rule.* For all $x : \sigma \, . \, C_\sigma(x) \Rightarrow C_\tau[A(x_\rho)/x_\rho, A(x_\psi)/x_\psi]$

Figure 4: Definition of the Subtype Relation (Constraint Rule)

about $\tau$ objects (which are in terms of values in $T$). The abstraction function $A$ is used to translate these values so that the predicates about $\tau$ objects make sense.

### 5.2.1 Discussion of Definition

The first clause addresses the need to relate values. It requires that abstraction functions *respect the invariant*: an abstraction function must map legal values of the subtype to legal values of the supertype. This requirement (and the assumption that the type specification preserves the invariant) suffices to argue that invariant properties of a supertype are preserved by the subtype.

The second clause addresses the need to relate non-extra methods of the subtype. Our formulation is similar to America's [1]. The first two signature rules are the standard contra/covariance rules. The exception rule says that $m_\sigma$ may not signal more than $m_\tau$, since a caller of a method on a supertype object should not expect to handle an unknown exception. The pre- and post-condition rules are the intuitive counterparts to the contravariant and covariant rules for signatures. 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 post-condition rule 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.

We do not consider invariants as shorthand for explicit conjuncts in a method's pre- and post-conditions because if we did the pre-condition rule would require that the supertype's invariant implies a subtype's. Usually just the opposite holds. For example, suppose a smallbag type is like the bag type except that its bound must be equal to 20:

**invariant** $\mid b_\rho.elems \mid \leq b_\rho.bound \wedge b_\rho.bound = 20$

To show smallbag is a subtype of bag, for the pre-condition rule for the equal method we would need to show that:

$I_{bag} \Rightarrow I_{smallbag}$

which is not true. In fact, the converse holds.

Finally, the third clause succinctly and directly states that constraints must be preserved. This requirement (and the assumption that each type specification satisfies its constraint) suffices to argue that history properties of a supertype are preserved.

### 5.2.2 Applying the Definition of Subtyping as a Checklist

Proofs of the subtype relation are usually obvious and can be done by inspection. Typically, the only interesting part is the definition of the abstraction function; the other parts of the proof are usually trivial. However, this section goes through the steps of an informal proof just to show what kind of reasoning is involved. Formal versions of these informal proofs are given in [28].

Let's revisit the stack and bag example using our definition as a checklist. Here $\sigma = < O_{stack}, S, \{push, pop, swap\_top, height, equal\} >$, and $\tau = < O_{bag}, B, \{put, get, card, equal\} >$. Recall that we represent a bounded bag's value as a pair, $< elems, bound >$, of a multiset of integers and a fixed bound, and a bounded stack's value as a pair, $< items, limit >$, of a sequence of integers and a fixed bound. It can easily be shown that each specification preserves its invariant and satisfies its constraint.

We use the abstraction function and the renaming map given in the specification for stack in Figure 3. The abstraction function states that for all $st : S$

$$A(st) = < mk\_elems(st.items), st.limit >$$

where the helping function, $mk\_elems : Seq \rightarrow M$, maps sequences to multisets and states that for all $sq : Seq,\ i : Int$:

$$mk\_elems([\,]) = \{\ \}$$
$$mk\_elems(sq \mid\mid [\ i\ ]) = mk\_elems(sq) \cup \{i\}$$

The renaming map $R$ is

$$R(push) = put$$
$$R(pop) = get$$
$$R(height) = card$$
$$R(equal) = equal$$

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

Next, we show the correspondences between *push* and *put*, between *pop* and *get*, etc. Let's look at the pre- and post-condition rules for just one method, *push*. Informally, the pre-condition rule for *put/push* requires that we show[4]:

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

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 stack's sequence component to the bag's 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.

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

Note 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 post-condition rule requires that we show *push*'s post-condition implies *put*'s. We can deal with the **modifies** and **ensures** parts separately. The **modifies** part holds because the same object is mentioned in both specifications. The **ensures** part follows from the definition of the abstraction function.

Finally, the constraint rule requires that we show that the constraint on stacks:

$$s_\rho.limit = s_\psi.limit$$

implies that on bags:

$$b_\rho.bound = b_\psi.bound$$

This is true because the length of the sequence component of a stack is the same as the size of the multiset component of its bag counterpart.

Note that we do not have to say anything specific for *swap_top*.

## 5.3 Second Definition: Extension Map

With the constraint approach users cannot use the history rule to deduce history properties. Our second approach allows them to do so. It requires that we "explain" each extra method in terms of existing methods. If such explanations are possible, the extra methods do not add any behavior that could not have been effected in their absence. Therefore, all supertype properties, including history properties, are preserved.

In our alternative definition, therefore, we do not add any constraints to our type specification (and thus remove the requirement that a type specification has to satisfy its constraint). Instead, to show that $\sigma$ is a subtype of $\tau$ we require a third mapping, which we call an *extension map*, that is defined for all extra methods introduced by the subtype. The extension map "explains" the behavior of each extra method as a program expressed in terms of non-extra methods. Interesting explanations are needed only for mutators; non-mutators always have the "empty" explanation, $\epsilon$.

Figure 5 gives the alternative definition. As before, we assume each type specification preserves its invariant. In defining the extension map, we intentionally leave unspecified the

DEFINITION OF THE SUBTYPE RELATION, <: $\sigma = < O_\sigma, S, M >$ is a *subtype* of $\tau = < O_\tau, T, N >$ if there exists an abstraction function, $A$, a renaming map, $R$, and an extension map, $E$, such that:

1. The abstraction function respects invariants:
   - *Invariant Rule.* $\forall s : S \;.\; I_\sigma(s) \Rightarrow I_\tau(A(s))$
2. Subtype methods preserve the supertype methods' behavior. If $m_\tau$ of $\tau$ is the corresponding renamed method $m_\sigma$ of $\sigma$, the following rules must hold:
   - *Signature rule.*
     - *Contravariance of arguments.* $m_\tau$ and $m_\sigma$ have the same number of arguments. If the list of argument types of $m_\tau$ is $\alpha_i$ and that of $m_\sigma$ is $\beta_i$, then $\forall i \;.\; \alpha_i < \beta_i$.
     - *Covariance of result.* Either both $m_\tau$ and $m_\sigma$ have a result or neither has. If there is a result, let $m_\tau$'s result type be $\gamma$ and $m_\sigma$'s be $\delta$. Then $\delta < \gamma$.
     - *Exception rule.* The exceptions signaled by $m_\sigma$ are contained in the set of exceptions signaled by $m_\tau$.
   - *Methods rule.* For all $x : \sigma$:
     - *Pre-condition rule.* $m_\tau.pre[A(x_{pre})/x_{pre}] = m_\sigma.pre$.
     - *Post-condition rule.* $m_\sigma.post \Rightarrow m_\tau.post[A(x_{pre})/x_{pre}, A(x_{post})/x_{post}]$
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 $\epsilon$.
   - *Extension rule.* For each new method, $m$, of $x : \sigma$, the following conditions must hold for $\pi$, the program to which $E(x.m(a))$ maps:
     - The input to $\pi$ is the sequence of objects $[x]||a$.
     - The set of methods invoked in $\pi$ is contained in the union of the set of methods of all types other than $\sigma$ 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 $\pi$. Let $\rho_1$ be the state in which both $m$ is invoked and $\pi$ starts. Assume $m.pre$ holds in $\rho_1$ and the call to $m$ terminates in state $\rho_2$. Then we require that $\pi$ terminates in state $\psi$ and

       $A(x_{\rho_2}) = A(x_\psi)$.

       Note that if $\pi = \epsilon, \psi = \rho_1$.

Figure 5: Definition of the Subtype Relation (Extension Rule)

language in which one writes a program, but imagine that it has the usual control structures, assignment, procedure call, etc.

### 5.3.1 Discussion of Definition

The first and second clauses are the same as in the first definition except that the pre-condition rule is stronger. Since the extension map is defined just for the extra methods, it is possible for a subtype to redefine a supertype's (non-extra) method in a way that causes a violation of a history property of the supertype. For example, suppose we have a window, $w$, with a *move* method

*move* = **proc** ($v$: vector)
  **requires** $v.x > 0 \wedge v.y > 0$
  **ensures** $w_{post}.center = w_{pre}.center + v$

that guarantees a window always moves in a northeasterly direction. Suppose a my_window type is just like window except with a weaker *move* method:

*move* = **proc** ($v$: vector)
  **ensures** $w_{post}.center = w_{pre}.center + v$

The methods rule given previously, in particular the pre-condition rule, holds, but clients of window objects would be surprised if a my_window object were used (and moved) in place of a window.[5] To rule out this behavior, we require that the pre-condition of each non-extra method be the same as the corresponding supertype's method.[6] Note that America uses the weaker pre-condition rule of Figure 4, and therefore he would erroneously allow subtype relations like this one, since his technique does not describe the constraints explicitly.

The third clause of the definition requires what is shown in the diamond diagram in Figure 6, read from top to bottom. We must show that the abstract value of the subtype object reached by running the extra method $m$ is also reached by running $m$'s explanation program. This diagram 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.

The extension rule constrains only what an explanation program does to its method's object, not to other objects. This makes sense because explanation programs do 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 ab-

[5]Thanks to Ian Maung for pointing out this problem and inspiring this example.

[6]An alternative solution to this problem would be to define the extension map for all methods, not just extra ones.

$$\begin{tikzcd}
& x_{\rho_1}:S \arrow[dl, "m"'] \arrow[dr, "E(x.m(a))"] & \\
x_{\rho_2}:S \arrow[dr, "A"'] & & x_{\psi}:S \arrow[dl, "A"] \\
& y:T &
\end{tikzcd}$$

Figure 6: The Diamond Diagram

stract values. We need only the weaker notion of *observable equivalence* (e.g., see Kapur's definition[20]), 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.

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. To show that history properties are preserved by non-extra mutators, we use the methods rule. However, because the properties are not stated explicitly, they cannot be proved for the extra methods. Instead extra methods must satisfy any possible property, which is surely guaranteed if the extra methods can be explained in terms of the non-extra methods via the extension map.

### 5.3.2 The Bag and Stack Example Again

The alternative definition of subtyping is also used as a checklist to prove a subtype relation. Besides the abstraction function, the only other interesting issue is the definition of the extension map. As was the case with the constraint approach, the actual proofs are usually trivial.

To prove that stack is a subtype of bag we follow the same procedure as in Section 5.2.2, except we need to show that the pre-conditions are identical, a trivial exercise for this example. We must additionally define an extension map to define *swap_top*'s effect. As stated earlier, it has the same effect as that described by the program, $\pi$, 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 equal abstract values. (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[19].) 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 $\rho_2$ is the termination state if we run *swap_top* and $\psi$ is the termination state if we run $\pi$. Therefore $A(s_{\rho_2}) = A(s_{\psi})$, since $A$ is a function.

## 5.4 Comparing the Two Definitions

The approach using explicit constraints is appealing because it is so simple. In addition, explicit constraints allow us to rule out unintended properties that happen to be true because of an error in a method specification. Having both the constraint and the method specifications is a form of useful redundancy: If the two are not consistent, this indicates an error in the specification. The error can then be removed (either by changing the constraint or some method specification). Therefore, including constraints in specifications makes for a more robust methodology.

Explicit constraints also allow us to state the common properties of type families directly. With the explanation approach, it is sometimes necessary to introduce extra methods in the supertype to ensure that history properties that do not hold for subtypes cannot be proved for supertypes. An example is given in Section 6, when we discuss a varying_bag type.

On the minus side is the loss of the history rule. Users are not permitted to use the history rule because if they did, they might be able to prove history properties that a subtype did not ensure. Therefore the specifier must be careful to define a strong enough constraint. In our experience the desired constraint is usually obvious. However, suppose the definer of fat_set mistakenly gives the following constraint:

**constraint** $| s_\rho | \leq | s_\psi |$

Users would then be unable to deduce that once an element is added to a fat_set it will always be there (since they are not allowed to use the history rule).

In summary, having an explicit constraint is appealing because the subtype relation is simple, it allows us to state properties of type families declaratively, and the constraint acts as a check on the correctness of a specification. The drawback is that if some property is left out of the constraint, there is no way users can make use of it.

## 6 Type Hierarchies

The requirement 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, 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 possibly additional "state." In the second, the subtype is more constrained than the supertype. We discuss these relationships below.

### 6.1 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 also 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) with methods to *insert* and *delete* elements, to *select* elements, and to provide the *size* of the set. A subtype, my_intset, might have more methods, e.g., *union*, *is_empty*. Here there is no extra state, just extra methods. If we are using the extension map approach, we must provide explanations for the extra methods, but for all but mutators, these are trivial. Thus, if *union* is a pure constructor, it has the empty explanation, $\epsilon$; otherwise it requires a non-trivial explanation, e.g., in terms of *insert.* If we are using the constraint approach, we must prove that the subtype's constraint implies that of the supertype. Often the two constraints will be identical, e.g., both intset and my_intset might have the trivial constraint.

Using either approach, it is easy to discover when a proposed subtype really is not one. For example, intset is not a subtype of fat_set because fat_sets only grow while intsets grow and shrink, i.e., it does not preserve various history properties of fat_set. If we are using the constraint approach, we would be unable to show that the intset contraint (which is trivial) implies that of fat_set; with the extension map approach, we will not be able to explain the effect of intset's *delete* method.

As a simple example of a many-to-one case, consider immutable pairs and triples (Figure 7). 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 because replacing the third element does not affect the first or second element. 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.

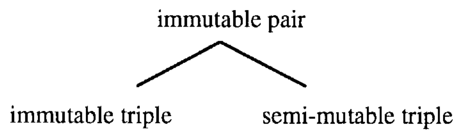

Figure 7: Pairs and Triples

Mutations of a subtype that would be visible through the methods of an immutable supertype are ruled out. For example, an immutable sequence, whose elements can 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 or because the constraint for sequences does not follow from that of arrays.

Many examples of extension subtypes are found in the literature. One common example concerns persons, employees, and students (Figure 8). 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.

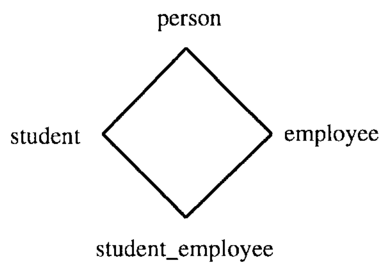

Figure 8: Person, Student, and Employee

Another example from the database literature concerns different kinds of ships [18]. The supertype is generic 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 [17]; 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 $\tau$ provides an *equal* method and consider a particular call *x.equal(y)*. The difficulty arises when x and y actually belong to $\sigma$, a subtype of $\tau$. 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** $result = (p.first = q.first \wedge p.second = q.second)$

(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** $result = (p.first = q.first \wedge p.second = q.second \wedge p.third = q.third)$

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** $result = (p.first = q.first \wedge p.second = q.second)$

*triple_equal* = **proc** ($p$: triple) **returns** (bool)
  **ensures** $result = (p.first = q.first \wedge p.second = q.second \wedge p.third = q.third)$

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.

## 6.2 Constrained Subtypes

The second type of subtype relation occurs when the subtype is more constrained than the supertype. 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, either in what the methods do or in the value spaces of objects or by having a tighter constraint. 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 9 shows the elephant hierarchy. The set of legal values for regular elephants includes all elephants whose color is grey or blue or white:

**invariant** $e_\rho.color = white \vee e_\rho.color = grey \vee e_\rho.color = blue$

The set of legal values for royal elephants is a subset of those for regular elephants:

**invariant** $e_\rho.color = blue$

and hence the abstraction function is into. The situation for albino elephants is similar. In addition, the *get_color* method for elephant is non-deterministic but deterministic for royal and albino elephants. This simple example has led others to define a subtyping relation that requires non-monotonic reasoning [25], 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.

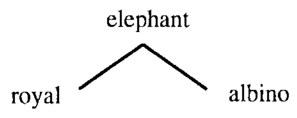

Figure 9: Elephant Hierarchy

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 10.

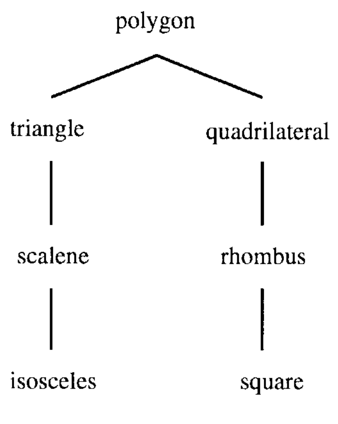

Figure 10: Polygon Hierarchy

The bag type discussed in Section 4.1 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 way different from *pop*.

In addition, since the actual value of the bound for bags was not defined, 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 (or constraint) that the bound of a bag does not change, that the call will succeed. A subtype of bag might limit the bound to a fixed value, or to a smaller range. Several subtypes of bag are shown in Figure 11; mediumbags have various bounds, so that this type might have its own subtypes, e.g., bag_150.

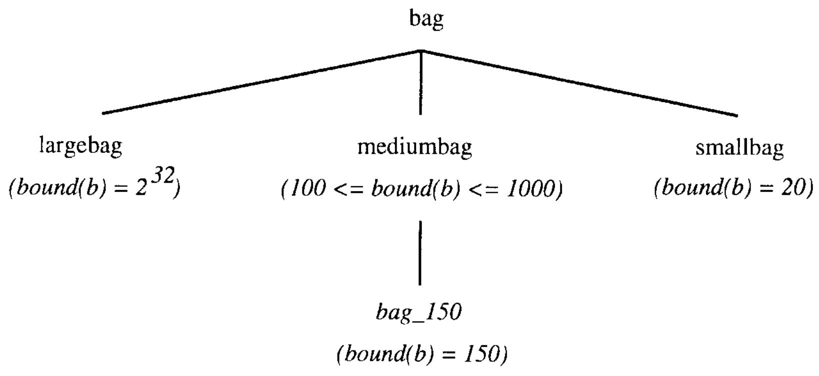

Figure 11: A Type Family for Bags

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 smallbag to be a subtype of largebag. However, the specifications for the two types are incompatible: the bound of every largebag is $2^{32}$, which is clearly not true for smallbags. 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 20, but the call is not legal for a smallbag. Therefore the pre-condition rule is not satisfied.

Although the bag type can have subtypes with different 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*:

```
$change\_bound$ = proc ($n$: int)
        requires $n \geq |b_{pre}.elems|$
        modifies $b$
        ensures $b_{post}.elems = b_{pre}.elems \wedge b_{post}.bound = n$
```

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 12 shows the new type hierarchy. This example points out an interesting difference between the two subtype definitions. If we are using the extension map approach, varying_bag would need to have a *change_bound* method that allows the bag's bound to change, but does not require it. The method is needed because otherwise the history rule would allow us to deduce that the bound does not change! The nondeterminism in its specification is resolved in its subtypes; bag (and its subtypes) provides 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), even though the method isn't interesting.

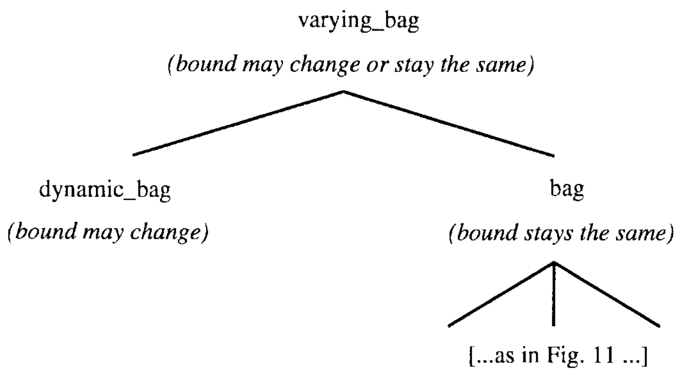

Figure 12: Another Type Family for Bags

On the other hand, if we are using the constraint approach, varying_bag and bag need not have a *change_bound* method. Instead, varying_bag simply has the trivial constraint. This means that its users cannot deduce anything about the bounds of its objects: the bound of an object might change or it might not. Therefore it can have both bag and dynamic_bag as subtypes. The constraint for bag (that a bag's bound does not change) allows users of its objects to depend on this property.

The varying_bag example illustrates a subtype that reduces nondeterminism in the constraint. The constraint for varying_bag can be thought of as being "either a bag's bound changes or it does not"; the constraint for bounded_bag reduces this nondeterminism by making a choice ("the bag's bound does not change"). A similar example is a family of integer counters shown in Figure 13. When a counter is advanced, we only know that its value gets bigger, so that the constraint is simply

**constraint** $c_\rho \leq c_\psi$

The doubler and multiplier subtypes have stronger constraints. For example, a multiplier's value always increases by a multiple, so that its constraint is:

**constraint** $\exists\ n : int\ .\ [\ n > 0 \wedge c_\rho = n * c_\psi\ ]$

For a family like this, we might choose to have an *advance* method for counter (so that each of its subtypes is constrained to have this method) or we might not, but this choice is available to us only if we use the constraint method.

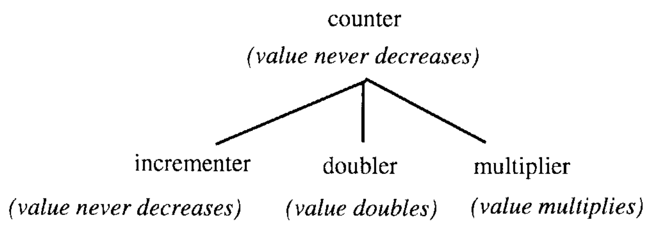

Figure 13: Type Family for Counters

In the case of the bag family illustrated in Figure 11, all types in the hierarchy might actually be implemented. However, sometimes supertypes are not intended to be implemented; instead they are *virtual types* that let us define the properties all 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. Also, larger integers cannot be a subtype of smaller ones because exceptions do not occur when expected. 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 14, 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.

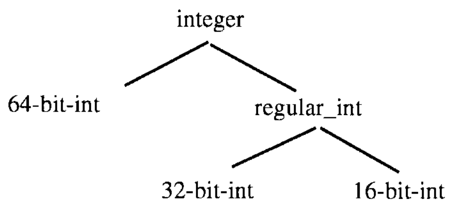

Figure 14: Integer Family

## 7 Related Work

Some of the research on defining subtype relations is concerned with capturing constraints on method signatures via the contra/covariance rules, such as those used in languages like Trellis/Owl [33], Emerald[3], Quest [5], Eiffel [30], POOL [1], and to a limited extent Modula-3 [32]. Our rules place constraints not just on the signatures of an object's methods, but also on their behavior.

Our work is most similar to that of America [2], who has proposed rules for determining based on type specifications whether one type is a subtype of another. (Meyer uses America's pre- and post-condition rules for Eiffel [30], although here the pre- and post-conditions are given "operationally," by providing a program to check them, rather than assertionally.) Cusack's approach [7] also relates specifications; her rule defines subtyping in terms of strengthening state invariants. However, neither author considers the problems introduced by extra mutators nor the preservation of history properties. Therefore, they allow certain subtype relations that we forbid (e.g., intset could be a subtype of fat_set in these approaches).

The emphasis on semantics of abstract types is a prominent feature of the work by Leavens. In his Ph.D. thesis [21] Leavens defines types in terms of algebras and subtyping in terms of a *simulation relation* between them. The work by Bruce and Wegner [4] is similar; like Leavens, they base their work on algebras. Leavens considered only immutable types. Dhara [10, 11, 23] extends Leavens' thesis work to deal with mutable types, but rules out the cases where extra methods cause problems; the rules are defined just for individual programs that have no aliasing between objects of related types, and therefore state changes caused by a subtype's extra methods cannot be observed through the supertype. Because of this restriction on aliasing they allow some subtype relations to hold where we do not. For example, they allow mutable pairs to be a subtype of immutable pairs whereas we do not.

In addition, these algebraic approaches are not constructive, i.e., they tell you what to look for, but not how to prove that you got it. Utting [36] does provide a constructive approach, but he bases his work in the refinement calculus language [31], a formalism that we believe is not very easy for programmers to deal with. Utting is not concerned with preserving history properties in the presence of extra methods and he also does not allow data refinement between supertype and subtype value spaces.

Others have worked on the specification of types and subtypes. For example, many have proposed Z as the basis of specifications of object types[8, 12, 6]; Goguen and Meseguer use FOOPS[15]; Leavens and his colleagues use Larch[22, 24, 11]. Though several of these researchers separate the specification of an object's creators from its other methods, none has identified the problem posed by the missing creators, and thus none has provided an explicit solution to this problem.

In summary, our work is similar in spirit to that of America and Cusack because they take a specification-based approach to defining a behavioral notion of subtyping. It complements the algebraic model-based approach taken by Leavens, Dhara, and Bruce and Wegner. Only America, Cusack, Utting, and Dhara deal with mutability, but none has addressed the need to preserve history properties. Only we have a technique that works in a general environment in which objects can be shared among possibly concurrent users.

## 8 Summary and Future Work

This paper defines a new notion of the subtype relation based on the semantic properties of the subtype and supertype. 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 require that a subtype preserve all the invariant and history properties of its supertype. We are particularly interested in an object's observable behavior (state changes), thus motivating our focus on history properties and on mutable types and mutators.

The paper presents two ways of defining the subtype relation, one using constraints and the other using the extension rule. Either of these approaches guarantees that subtypes preserve their supertypes' invariant and history properties. Ours is the first work to deal with history properties, and to provide a way of determining the acceptability of the "extra" methods in the presence of mutability.

The paper also presents a way to specify the semantic properties of types formally. One reason we chose to base our approach on Larch is that Larch allows formal proofs to be done entirely in terms of specifications. In fact, once the theorems corresponding to our subtyping rules are formally stated in Larch, their proofs are almost completely mechanical—a matter of symbol manipulation—and could be done with the assistance of the Larch Prover[14].

Although we gave two formal definitions of the subtype relation, we did not formally characterize the criterion against which we can measure the soundness of our definitions. We only argued informally that our definitions guarantee that a subtype's objects behave the same, e.g., preserve properties, as their supertype's. A formal characterization of this criterion remains another open research problem. One possibility is to do this within the Larch framework. In Larch, the meaning of a specification is the theory derived from a set of axioms and rules. A possible correctness criterion is to require the theory of a subtype to contain those of its supertypes.

In developing our definitions, 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 that is 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 approaches relatively easy to apply and expect them to be used primarily in an informal way. The essence of a subtype relationship (in either of our approaches) 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 can be done at this point also; they are usually trivial and can be done by inspection.

**Acknowledgments**

Special thanks to John Reynolds who provided perspective and insight that led us to explore alternative definitions of subtyping and their effect on our specifications. We thank Gary

Leavens for a helpful discussion on subtyping and pointers to related work. In addition, Gary, John Guttag, Greg Morrisett, Bill Weihl, Eliot Moss, Amy Moormann Zaremski, Mark Day, Sanjay Ghemawat, and Deborah Hwang gave useful comments on earlier versions of this paper.

## References

[1] Pierre America. A parallel object-oriented language with inheritance and subtyping. *SIGPLAN*, 25(10):161–168, October 1990.

[2] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *LNCS*, pages 60–90. Springer-Verlag, NY, 1991.

[3] Andrew P. Black, Norman Hutchinson, Eric Jul, Henry M. Levy, and Larry Carter. Distribution and abstract types in emerald. *IEEE TSE*, pages 65–76, January 1987.

[4] K.B. Bruce and P. Wegner. An algebraic model of subtypes in object-oriented languages (draft). In *ACM SIGPLAN Notices*, volume 21, October 1986. Object-Oriented Programming Workshop.

[5] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[6] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, , and P. Smith. Object-z: An object oriented extension to z. In *FORTE89, International Conference on Formal Description Techniques*, December 1989.

[7] Elspeth Cusack. Inheritance in object oriented z. In *Proceedings of ECOOP '91*, 1991.

[8] Elspeth Cusack and Michael Lai. Object-oriented specification in lotos and z, or my cat really is object-oriented! In J. W. de Bakker, W. P de Roever, and G. Rozenberg, editors, *Foundations of Object Oriented Languages*, pages 179–202. Springer Verlag, June 1991. LNCS 489.

[9] O-J. Dahl, B. Myrhaug, and K. Nygaard. SIMULA common base language. Technical Report 22, Norwegian Computing Center, Oslo, Norway, 1970.

[10] Krishna Kishore Dhara. Subtyping among mutable types in object-oriented programming languages. Master's thesis, Iowa State University, Ames, Iowa, 1992.

[11] Krishna Kishore Dhara and Gary T. Leavens. Subtyping for mutable types in object-oriented programming languages. Technical Report 92-36, Department of Computer Science, Iowa State University, Ames, Iowa, November 1992.

[12] David Duke and Roger Duke. A history model for classes in object-z. In *Proceedings of VDM '90: VDM and Z*, 1990.

[13] B. Liskov et al. *CLU Reference Manual*. Springer-Verlag, 1981.

[14] S.J. Garland and J.V. Guttag. An overview of LP, the Larch Prover. In *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 137–151, Chapel Hill, NC, April 1989. Lecture Notes in Computer Science 355.

[15] Joseph A. Goguen and Jose Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object Oriented Programming.* MIT Press, 1987.

[16] John V. Guttag, James J. Horning, and Jeannette M. Wing. The larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[17] Daniel C. Halbert and Patrick D. O'Brien. Using types and inheritance in object-oriented programming. *IEEE Software*, pages 71–79, September 1987.

[18] M Hammer and D. McLeod. A semantic database model. *ACM Trans. Database Systems*, 6(3):351–386, 1981.

[19] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(1):271–281, 1972.

[20] Deepak Kapur. Towards a theory of abstract data types. Technical Report 237, MIT LCS, June 1980. Ph.D. thesis.

[21] Gary Leavens. Verifying object-oriented prograsm that use subtypes. Technical Report 439, MIT Lab. for Computer Science, February 1989. Ph.D. thesis.

[22] Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[23] Gary T. Leavens and Krishna Kishore Dhara. A foundation for the model theory of abstract data types with mutation and aliasing (preliminary version). Technical Report 92-35, Department of Computer Science, Iowa State University, Ames, Iowa, November 1992.

[24] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes. In *ECOOP/OOPSLA '90 Proceedings*, 1990.

[25] Udo Lipeck. Semantics and usage of defaults in specifications. In *Foundations of Information Systems Specification and Design*, March 1992. Dagstuhl Seminar 9212 Report 35.

[26] B. Liskov. Preliminary design of the thor object-oriented database system. In *Proc. of the Software Technology Conference.* DARPA, April 1992. Also Programming Methodology Group Memo 74, MIT Laboratory for Computer Science, Cambridge, MA, March 1992.

[27] B. Liskov and J. Guttag. *Abstraction and Specification in Program Design.* MIT Press, 1985.

[28] B.H. Liskov and J.M. Wing. Family values: A semantic notion of subtyping. Technical Report 562, MIT Lab. for Computer Science, 1992. Also available as CMU-CS-92-220.

[29] David Maier and Jacob Stein. Development and implementation of an object-oriented dbms. In S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 167–185. Morgan Kaufmann, 1990.

[30] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, 1988.

[31] Carroll Morgan. *Programming from Specifications.* Prentice Hall, 1990.

[32] Greg Nelson. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[33] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis: Object-based environment language reference manual. Technical Report 372, Digital Equipment Corp./Easter Research Lab., 1985.

[34] John Scheid and Steven Holtsberg. Ina jo specification language reference manual. Technical Report TM-6021/001/06, Paramax Systems Corporation, A Unisys Company, June 1992.

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

[36] Mark Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning.* PhD thesis, University of New South Wales, Australia, 1992.