Behavioral Subtyping Using Invariants and Constraints

Barbara H. Liskov[a] Jeannette M. Wing

July 1999
CMU-CS-99-156

[a]MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

The work described in this paper is based on a November 1994 *ACM TOPLAS* paper, "A Behavioral Notion of Subtyping, by the same authors. This paper's version has been submitted to the volume *Formal Methods For Distributed Processing: An Object-Oriented Approach*, edited by Howard Bowman and John Derrick.

Abstract

We present a way of defining the subtype relation that ensures that subtype objects preserve behavioral properties of their supertypes. The subtype relation is based on the specifications of the sub- and supertypes. Our approach handles mutable types and allows subtypes to have more methods than their supertypes. Dealing with mutable types and subtypes that extend their supertypes has surprising consequences on how to specify and reason about objects. In our approach, we discard the standard data type induction rule, we prohibit the use of an analogous "history" rule, and we make up for both losses by adding explicit predicates—invariants and constraints–to our type specifications. We also discuss the ramifications of our approach of subtyping the design of type families.

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.

Keywords: subtype, object-oriented design, abstraction function, mutable types, invariants, constraints, specifications

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[DMN70], C++[Str86], Modula-3[Nel91], and Trellis/Owl[SCB+86], 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[BHJ+87, SCB+86, Car88]. 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:

> *Subtype Requirement*: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

A type's specification determines what properties we can prove about objects.

We are interested only in *safety* properties ("nothing bad happens"). First, properties of an object's behavior in a particular program must be preserved: to ensure that a program continues to work as expected, calls of methods made in the program that assume the object belongs to a supertype must have the same behavior when the object actually belongs to a subtype. In addition, however, properties independent of particular programs must be preserved because these are important when independent programs share objects. We focus on two kinds of such properties: *invariants*, which are properties true of all states, and *history properties*, which are properties true of all sequences of states. We formulate invariants as predicates over single states and history properties, over pairs 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 the bag's bound does not change. We do not address other kinds of safety properties of computations, e.g., the existence of an object in a state, the number of objects in a state, or the relationship between objects in a state, since these do not have to do with the meanings of types. We also do not address *liveness* properties ("something good eventually happens"), e.g., the size of a bag will eventually reach the bound.

This chapter provides a general, yet easy to use, definition of the subtype relation that satisfies the Subtype Requirement. Our approach handles mutable types and allows subtypes to have more methods than their supertypes. Dealing with mutable types and subtypes that extend their supertypes has surprising consequences on how to specify and reason about objects. In our approach, we discard the standard data type induction rule, we prohibit the use of an analogous "history" rule, and we make up for both losses by adding explicit predicates to our type specifications. 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 [LG85].

Our definition applies in a very general distributed 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.

The chapter also 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 and thus allow generic routines to be written that work for all family members. Our approach makes it particularly easy to define type families: it emphasizes the properties that all family members must preserve, and it does not require the introduction of unnecessary methods (i.e., methods that the supertype would not naturally have).

The chapter is organized as follows. Section 2 discusses in more detail what we require of our subtype relation and provides the motivation for our approach. We describe our model of computation in Section 3 and present our specification method in Section 4. We give a formal definition of subtyping in Section 5; we discuss its ramifications on designing type hierarchies in Section 6. We describe related work in Section 7 and summarize our contributions in Section 8.

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 kinds of collections behave similarly. The main difference is that 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 [Hoa72]. 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 *push* method and bag's *put* and similarly for the *pop* and *get* 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 programs 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 or processes. In particular, we need to pay attention to extra *mutator* methods (like *swap_top*) that modify their object.

Consider first the case of aliasing. The problem here is that within a program 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 σ is a subtype of τ and that variables

x: τ
y: σ

both denote the same object (which must, of course, belong to σ or one of its subtypes). When the object is accessed through x, only τ methods can be called. However, when it is used through y, σ methods can be called and if these methods are mutators, their effects will be visible later when the object is accessed via

x. To reason about the use of variable x using the specification of its type τ, 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 [Lis92] and Gemstone [MS90]). 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 a 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 know only about the "apparent" type of the object; the subtype ought to preserve any properties that can be proved about the supertype. In particular, 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.

Our approach achieves this goal by adding information to type specifications. To handle invariants, we add an **invariant** clause; to handle history properties, a **constraint** clause. Showing that σ is a subtype of τ requires showing that (under the abstraction function) σ's invariant implies τ's invariant and σ's constraint implies τ's constraint.

For example, for the bag and stack example, the two invariants are identical: both state that the size of the bag (stack) is less than or equal to its bound. Similarly, the two constraints are identical: both state that the bound of the bag (or stack) does not change. Showing that stack's invariant and constraint respectively imply bag's invariant and constraint is trivial. The extra method *swap_top* is permitted because even though it changes the stack's contents, it preserves stack's invariant and constraint.

In Section 5 we present and discuss our subtype definition. 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. Effectively *Obj* is a set of unique identifiers for all objects that can contain values.

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, ρ, with an environment, $\rho.e$, and store, $\rho.s$, we use the notation x_ρ to denote the value of x in state ρ; 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, $\langle O, V, M\rangle$, 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 *producer*, an *observer*, or a *mutator*. Producers of an object of type τ return new objects of type τ; observers return results of other types; mutators modify objects of type τ. An object is *immutable* if its value cannot change and otherwise it is *mutable*; a type is immutable if its objects are and otherwise it is mutable. Clearly a type can be mutable only if some of its methods are mutators. We allow *mixed methods* where a producer 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*. (These are often called *constructors* in the literature.) Unlike other kinds of methods, creators do not belong to particular objects, but rather are independent operations.

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

$$\rho_0 \ \ Tr_1 \ \ \rho_1 \ \ \ldots \ \ \rho_{n-1} \ \ Tr_n \ \ \rho_n$$

Each transition, Tr_i, of a computation sequence is a partial function on states; we assume the execution of each transition is atomic. A *history* is the subsequence of states of a computation; we use ρ and ψ to range over states in any computation, c, where ρ precedes ψ in c. The value of an object can change only through the invocation of a mutator; in addition the environment can change through assignment and the domain of the store can change through the invocation of a creator or producer.

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;
- A definition of the type's invariant and history properties;
- For each of the type's methods:
  - Its name;
  - Its signature (including signaled exceptions);
  - Its behavior in terms of pre-conditions and post-conditions.

Note that the creators are missing. Omitting creators allows subtypes to provide different creators than their supertypes. In addition, omitting creators makes it easy for a type to have multiple implementations, allows new creators to be added later, and reflects common usage: for example, Java interfaces and virtual types provide no way for users to create objects of the type. We show how to specify creators in Section 4.2.

In our work we use formal specifications in the two-tiered style of Larch [GHW85]. 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, $\langle elems, bound \rangle$, 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, $\in$ is the membership operation, and $\mid x \mid$ is a cardinality operation that returns the total number of elements in the multiset x. These operations as well as equality ($=$) and inequality ($\neq$) are all defined in BBag.

The **invariant** clause contains a single-state predicate that defines the type's invariant properties. The **constraint** clause contains a two-state predicate that defines the type's history properties. We will discuss these clauses in more detail in subsequent sections.

---

bag = **type**

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

invariant $| \; b_\rho.elems \; | \leq b_\rho.bound$
constraint $b_\rho.bound = b_\psi.bound$

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

get = **proc** ( ) **returns** (int)
  requires $b_{pre}.elems \neq \{\}$
  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

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. We use *result* to name a method's result parameter. 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.[1] 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 the same object.

A method m's *pre-condition*, denoted $m.pre$, is the predicate that appears in its **requires** clause; e.g., *put*'s pre-condition checks to see that adding an element will not enlarge the bag beyond its bound. If the clause is missing, the pre-condition is trivially "true."

A method m's *post-condition*, denoted $m.post$, is the conjunction of the predicates given by its **modifies** and **ensures** clauses. A **modifies** $x_1, \ldots, x_n$ clause is shorthand for the predicate:

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

[1] Note that pre and $post$ are implicitly universally quantified variables over states. Also, more formally, x_{pre} stands for $pre.s(pre.e(x))$; x_{post} stands for $post.s(post.e(x))$.

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. For example, *card*'s post-condition says that it returns the size of the bag and no objects (including the bag) change, and *put*'s post-condition says that the bag's value changes by the addition of its integer argument, and no other objects change.

Methods may terminate normally or exceptionally; the exceptions are listed in a **signals** clause in the method's header. For example, instead of the *get* method we might have had

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

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 $\mathbf{new}(x)$ stands for the predicate:

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

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

---

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

bag_create_small = **proc** ( ) **returns** (bag)
  ensures **new**(*result*) $\wedge\ result_{post} = \langle\{\}, 100\rangle$

bag_create_single = **proc** (i: int) **returns** (bag)
  ensures **new**(*result*) $\wedge\ result_{post} = \langle\{i\}, 100\rangle$

Figure 2: Creator Specifications for Bags

4.3 Type Specifications Need Explicit Invariants

By not including creators in type specifications and by allowing subtypes to extend supertypes with mutators 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 (in particular each mutator) preserve the invariant. Without the creators, we have no base case. Without knowing all mutators of type τ (as added by τ's subtypes), we have an incomplete inductive case. With no data type induction rule, we cannot prove type invariants!

To compensate for the lack of a data type induction rule, we state the invariant explicitly in the type specification through 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 τ. For example, we include

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

in 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 τ stands for the predicate: For all computations, c, and all states ρ in c,

$\forall x : \tau \,.\, x \in dom(\rho) \Rightarrow \phi(x_\rho)$

Any additional invariant property 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.

As part of specifying a type and its creators we must show that the invariant holds for all objects of the type. All creators for a type τ must *establish* τ*'s invariant*, I_τ:

For each creator for type τ, show for all $x\!:\!\tau$ that $I_\tau[result_{post}/x_\rho]$.

where $P[a/b]$ stands for predicate P with every occurrence of b replaced by a. Similarly, each producer must establish the invariant on its newly-created object. In addition, each mutator of the type must *preserve the invariant*. To prove this, we assume each mutator is called on an object of type τ with a legal value (one that satisfies the invariant), and show that any value of a τ object it modifies is legal:

For each mutator m of τ, for all $x\!:\!\tau$ assume $I_\tau[x_{pre}/x_\rho]$ and show $I_\tau[x_{post}/x_\rho]$.

For example, we would need to show that the three creators for bag establish the invariant, and that *put* and *get* preserve the invariant for bag. (We can ignore *card* and *equal* because they are observers.) Informally the invariant holds because each creator guarantees that the size is no larger than the bound; *put*'s pre-condition checks that there is enough room in the bag for another element; and *get* either decreases the size of the bag or leaves it the same.

The loss of data type induction means that additional invariants cannot be proved. Therefore the specifier must be careful to define an invariant that is strong enough that all desired invariants follow from it.

4.4 Type Specifications Need Explicit Constraints

We are interested in the history properties of objects in addition to their invariant properties. We can formulate history properties as predicates over state pairs, and prove them using the *history rule*:

History Rule: For each of the i mutators m of τ, for all $x : \tau$:

$$\frac{m_i.pre \wedge m_i.post \Rightarrow \phi[x_{pre}/x_\rho, x_{post}/x_\psi]}{\phi(x_\rho, x_\psi)}$$

We cannot use this history rule directly, however. It is incomplete since subtypes may define additional mutators. If we use it without considering the extra mutators, it is easy to prove properties that do not hold for subtype objects!

To compensate for the lack of the history rule, we state history properties explicitly in the type specification through a **constraint** clause[2]; if the constraint is trivial, the clause can be omitted. For example, the constraint

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

in the specification of bag declares that a bag's bound never changes. 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 $\forall\, i : int \,.\, i \in s_\rho \Rightarrow i \in s_\psi$

The predicate $\phi(x_\rho, x_\psi)$ appearing in a **constraint** clause for type τ stands for the predicate: For all computations, c, and all states ρ and ψ in c such that ρ precedes ψ,

$\forall x : \tau \,.\, x \in dom(\rho) \Rightarrow \phi(x_\rho, x_\psi)$

[2] The use of the term "constraint" is borrowed from the Ina Jo specification language [SH92], which also includes constraints in specifications.

Note that we do not require that ψ be the immediate successor of ρ in c.

Just as we had to prove that methods preserve the invariant, we must show that they *satisfy the constraint.* This is done by using the history rule for each mutator.

The loss of the history rule is analogous to the loss of a data type induction rule. A practical consequence of not having a history rule is that the specifier must make the constraint strong enough so that all desired history properties follow from it.

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, $\langle items, limit \rangle$, 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. The constraint indicates that the stack's limit does not change. 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.

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 5.3.) The **subtype** clause also lets specifiers relate subtype methods to those of the supertype. The subtype must provide all methods of its supertype; we refer to these as the *inherited* methods.[3] Inherited methods can be renamed, e.g., *push* for *put*; all other methods of the supertype are inherited without renaming, e.g., *equal*. In addition to the inherited methods, the subtype may also have some *extra* methods, e.g., *swap_top*. (Stack's *equal* method must take a bag as an argument to satisfy the contravariance requirement. We discuss this issue further in Section 6.1.)

5.2 Definition of Subtype

The formal definition of the subtype relation, $\preceq$, is given in Figure 4. It relates two types, σ and τ, each of whose specifications respectively preserves its invariant, I_σ and I_τ, and satisfies its constraint, C_σ and C_τ. In the rules, 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 predicates about τ objects (which are in terms of values in T). The abstraction function A is used to translate these values so that the predicates about τ objects make sense. A may be partial, need not be onto, but can be many-to-one. We require that an abstraction function be defined for all legal values of the subtype (although it need not be defined for values that do not satisfy the subtype invariant). Moreover, it must map legal values of the subtype to legal values of the supertype.

The first clause addresses the need to relate inherited methods of the subtype. Our formulation is similar to America's [Ame90]. The first two signature rules are the standard contra/covariance rules. 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- 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 at least in any state required by the supertype. 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.

The second clause addresses preserving program-independent properties. The invariant rule 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 argument for the preservation of subtype's history properties

[3] We do not mean that the subtype inherits the code of these methods but simply that it provides methods with the same behavior (as defined below) as the corresponding supertype methods.

stack = **type**

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

invariant $length(s_\rho.items) \leq s_\rho.limit$
constraint $s_\rho.limit = s_\psi.limit$

$push$ = **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: bag) **returns** (bool)
    ensures $result = (s = t)$

subtype of bag ($push$ **for** put, pop **for** get, $height$ **for** $card$)
$\forall st : S \ .\ A(st) = \langle mk\_elems(st.items), st.limit \rangle$
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

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

1. Subtype methods preserve the supertype methods' behavior. 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 $\forall i \,.\, \alpha_i \preceq \beta_i$.
     - *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 α and m_σ's be β. Then $\beta \preceq \alpha$.
     - *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.* $m_\tau.pre[A(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre$.
     - *Post-condition rule.* $m_\sigma.post \Rightarrow m_\tau.post[A(x_{pre})/x_{pre}, A(x_{post})/x_{post}]$
2. Subtypes preserve supertype properties. For all computations, c, and all states ρ and ψ in c such that ρ precedes ψ, for all $x : \sigma$:
   - *Invariant Rule.* Subtype invariants ensure supertype invariants.

     $I_\sigma \Rightarrow I_\tau[A(x_\rho)/x_\rho]$
   - *Constraint Rule.* Subtype constraints ensure supertype constraints.

     $C_\sigma \Rightarrow C_\tau[A(x_\rho)/x_\rho, A(x_\psi)/x_\psi]$

Figure 4: Definition of the Subtype Relation

is completely analogous, using the constraint rule and the assumption that the type specification satisfies its constraint.

We do not include the invariant in the methods (or constraint) rule directly. For example, the pre-condition rule could have been

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

We omit adding the invariant because if it is needed in doing a proof it can always be assumed, since it is known to be true for all objects of its type.

Note that in the various rules we require $x : \sigma$, yet x appears in predicates concerning τ objects as well. This makes sense because $\sigma \preceq \tau$.

5.3 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 straightforward. 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 [LW92].

Let's revisit the stack and bag example using our definition as a checklist. Here

$$\sigma = \langle O_{stack}, S, \{push, pop, swap\_top, height, equal\}\rangle$$
$$\tau = \langle O_{bag}, B, \{put,\ get,\ card, equal\}\rangle$$

Recall that we represent a bounded bag's value as a pair, $\langle elems, bound\rangle$, of a multiset of integers and a fixed bound, and a bounded stack's value as a pair, $\langle items, limit\rangle$, 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 $st : S$

$$A(st) = \langle mk\_elems(st.items), st.limit\rangle$$

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

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

A is partial; it is defined only for sequence–natural numbers pairs, $\langle items, limit\rangle$, where $limit$ is greater than or equal to the size of $items$.

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 $\mid A(s_{pre}).elems \mid < A(s_{pre}).bound$.

4. *push*'s pre-condition holds by substituting equals for equals.

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.

The invariant rule requires that we show that the invariant on stacks:

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

implies that on bags:

$$\mid A(s_\rho).elems \mid \leq A(s_\rho).bound$$

[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 (b appears in the bag specification).

We can show this by a simple proof of induction on the length of the sequence of a bounded stack.

The constraint rule requires that we show that the constraint on stacks:

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

implies that on bags:

$$A(s_\rho).bound = A(s_\psi).bound$$

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*; it is taken care of just like all the other methods when we show that the specification of stack satisfies its invariant and constraint.

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 looked at 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. In practice, many type families will exhibit both kinds of relationships.

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, intset2, might have more methods, e.g., *union*, *is_empty*. Here there is no extra state, just extra methods. Suppose intset's invariant and constraints are both trivial; intset2's would be as well. Thus, proving that intset2 preserves intset's invariant and constraint is trivial.

It is easy to discover when a proposed subtype really is not one. For example, the fat_set type discussed earlier has an *insert* method but no *delete* method. Intset is not a subtype of fat_set because fat_sets only grow while intsets grow and shrink; intset does not preserve various history properties of fat_set, in particular, the constraint that once some integer is in the fat_set, it remains in the fat_set. The attempt to show that the intset constraint (which is trivial) implies that of fat_set would fail.

As a simple example of a many-to-one case, consider immutable pairs and triples (Figure 5). 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.

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 the constraint for sequences does not follow from that for arrays.

Many examples of extension subtypes are found in the literature. One common example concerns persons, employees, and students (Figure 6). 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

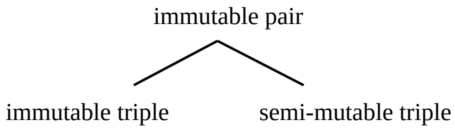

Figure 5: Pairs and Triples

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.

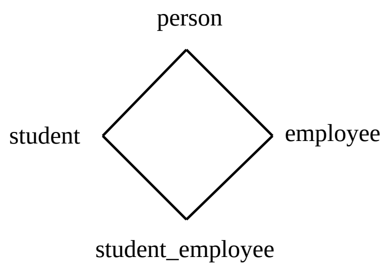

Figure 6: Person, Student, and Employee

Another example from the database literature concerns different kinds of ships [HM81]. 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 [HO87]; 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 $x = \langle 0, 0, 0 \rangle$ and $y = \langle 0, 0, 1 \rangle$. 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, we would expect two triples to be equal only if their first, second, and third components were equal. 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," but 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. (In some object-oriented languages, such as Java, the additional equal methods are obtained by overloading.)

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 kind of subtype relation occurs when the subtype is more constrained than the supertype. In this case, the supertype specification is written in a way that allows variation in behavior among its subtypes. Subtypes constrain the supertype by reducing the variability. 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.

Since constrained subtypes reduce variation, it is crucial when defining this kind of type hierarchy to think carefully about what variability is permitted for the subtypes. The variability will show up in the supertype specifications in two ways: in the invariant and constraint, and also in the specifications of the individual methods. In both cases the supertype definitions will be nondeterministic in those places where different subtypes are expected to provide different behavior.

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 7 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. Furthermore, the elephant method that returns the color (if there is such a method) can return grey or blue or white, i.e., it is nondeterministic; the subtypes restrict the nondeterminism for this method by defining it to return a specifc color.

This simple example has led others to define a subtyping relation that requires non-monotonic reasoning [Lip92], but we believe it is better to use variability in the supertype 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 the specification of the supertype.

The bag type discussed in Section 4.1 has two kinds of variability. First, 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, the actual value of the bound for bags is not defined; it can be any natural number, thus allowing subtypes to have different bounds. This variability shows up in the specification of *put*, where we

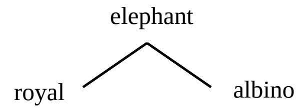

Figure 7: Elephant Hierarchy

do not say what specific bound value causes the call to fail. Therefore, a user of *put* must be prepared for a failure. (Of course the user could deduce that a particular call will succeed, based on a previous sequence of method calls and the constraint that the bound of a bag does not change.) 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 8; mediumbags have various bounds, so that this type might have its own subtypes, e.g., bag_150.

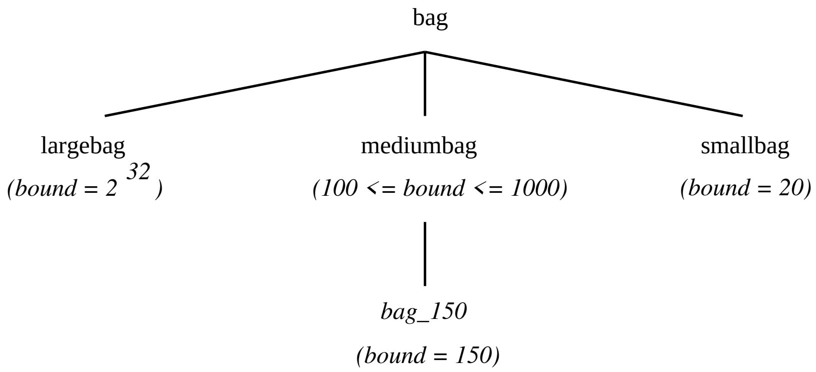

Figure 8: A Type Family for Bags

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 cannot have subtypes where the bounds of the bags can change dynamically. If we wanted a type family that included both bag and such dynamic bags, we would need to define a supertype in which the bound is allowed, but not required, to vary. Figure 9 shows the new type hierarchy. Dynamic_bags have a bound that tracks the size: each time an element is added or removed from a dynamic_bag, the bound changes to match the new size. Flexible_bags have an additional mutator, *change_bound*:

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

Notice that other types in the family need not have a *change_bound* method.

This example illustrates the different ways that subtypes reduce variability. All varying_bag subtypes reduce variability in the specification for the *put* method; varying_bag's *put* method is non-deterministic,

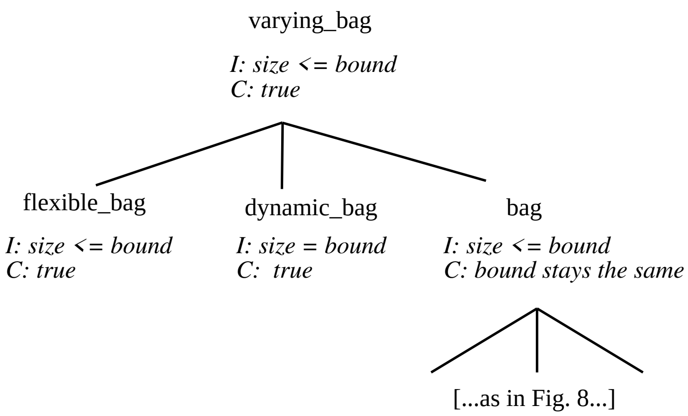

Figure 9: Another Type Family for Bags

since it might add the element (and change the bound) if the size is the same as the bound, or it might not. Bag and flexible_bag reduce this variability by not adding the element, whereas dynamic_bag does add the element. In addition, bag reduces variability by restricting the constraint: the trivial constraint for varying_bag can be thought of as stating "either a bag's bound may change or it stays the same;" the constraint for bag reduces this variability by making a choice ("the bag's bound stays the same") and users can then rely on this property for bags and its subtypes. Dynamic_bag reduces variability by restricting varying_bag's invariant so that it no longer allows the size to be less than the bound. Finally, flexible_bag reduces variability because of the extra mutator, *change_bound*; all its subtypes must allow explicit re-setting of the bound.

Another example is a family of integer counters shown in Figure 10. 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. If we do provide an advance method, its specification will have to be nondeterministic (i.e., it merely states the the size of the counter grows) to allow the subtypes to provide the definitions that are appropriate for them.

In the case of the bag family illustrated in Figure 8, all types in the hierarchy might be "real" in the sense that they have objects. However, sometimes supertypes are *virtual*; they define the properties all subtypes have in common but have no objects of their own. Varying_bag of Figure 9 might be such a type.

Virtual types are useful in many type hierarchies. For example, we would use them to 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 virtual supertype that includes them. Such a hierarchy is shown in Figure 11, where integer is a virtual type whose invariant simply says that the size of an integer is greater than zero. Integer types with different sizes are subtypes of integer. In addition, small integer types are subtypes of regular_int, another virtual type; the invariant in the specification for regular_int states that the size of an integer is either 16 bits or 32 bits. An integer family might have a structure like this, or it might be flatter by having all integer types be direct subtypes of integer.

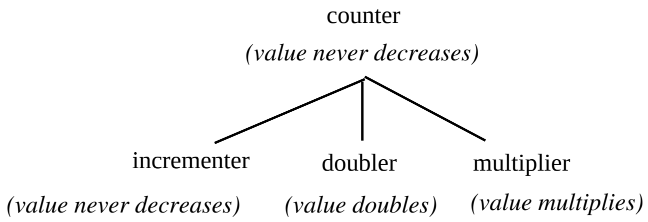

Figure 10: Type Family for Counters

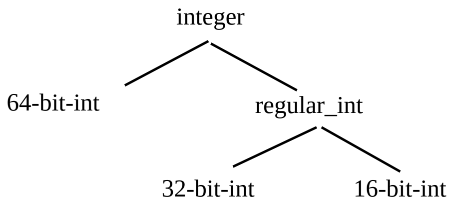

Figure 11: Integer Family

7 Related Work

Some 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 [SCB+86], Emerald[BHJ+87], Quest [Car88], Eiffel [Mey88], POOL [Ame90], and to a limited extent Modula-3 [Nel91]. 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 [Ame91], who has proposed rules for determining based on type specifications whether one type is a subtype of another. Meyer [Mey88] also uses pre- and post-condition rules similar to America's and ours. Cusack's [Cus91] approach of relating type specifications defines subtyping in terms of strengthening state invariants. However, none of these authors considers neither 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).

Our use of constraints in place of the history rule is one of two techniques discussed in [LW94]. That paper proposes a second technique in which there is no constraint; instead, extra methods are not allowed to introduce new behavior. It requires that the behavior of each extra mutator be "explained" in terms of existing behavior, through existing methods. We believe the use of constraints is simpler and easier to reason about than this "explanation" approach.

The emphasis on semantics of abstract types is a prominent feature of the work by Leavens. In his Ph.D. thesis Leavens [Lea89] defines types in terms of algebras and subtyping in terms of a *simulation relation* between them. His simulation relations are a more general form of our abstraction functions. Leavens considered only immutable types. Dhara [Dha92, DL92, LD92] extends Leavens' thesis work to deal with mutable types, but rules out the cases where extra methods cause problems, e.g., aliasing. Because of their restrictions 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.

Others have worked on the specification of types and subtypes. For example, many have proposed Z as the basis of specifications of object types[CL91, DD90, CDD+89]; Goguen and Meseguer[GM87] use FOOPS; Leavens and his colleagues use Larch[Lea91, LW90, DL92]. 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.

8 Summary

We defined 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 the behavior of the supertype methods and also all 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.

We also presented 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[GG89, ZW97].

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 that is needed to ensure that programs that work correctly using the supertype continue to work correctly with the subtype.

Programmers have found our approach relatively easy to apply and use it primarily in an informal way. The essence of a subtype relationship is expressed in the mappings. These mappings can be defined informally, 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 also be done informally, in the style given in Section 5.3; they are usually straightforward and can be done by inspection.

We also showed that our approach is useful by looking at a number of examples. This led us to identify two kinds of subtypes: ones that extend the supertype, and ones that constrain it. In the former case, the supertype can be defined without a great deal of thought about the subtypes, but in the latter case, this is not possible; instead the supertype specification must be done carefully so that it allows all of the intended subtypes. In particular the specification of the supertype must contain sufficient nondeterminism in the invariant, constraint, and method specifications.

Our analysis raises two issues about type hierarchy that have been ignored previously by both the formal methods and object-oriented communities. First, subtypes can have more methods, specifically more mutators, than their supertypes. Second, subtypes need to have different creators than supertypes. These issues forced us to revisit proof rules normally associated with type specifications: the data type induction rule and the history rule. We decided to preclude the use of these rules, and to have explicit invariants and constraints to replace them. Although it is possible to define a subtype relation that avoids explicit invariants and constraints, doing so is awkward and often requires invention of superfluous supertype methods and creators. We prefer to use explicit invariants and constraints because this allows a more direct way of capturing the designer's intent.

References

[Ame90] America, P. A parallel object-oriented language with inheritance and subtyping. *SIGPLAN*, 25(10):161–168, October 1990.

[Ame91] America, P. Designing an object-oriented programming language with behavioural subtyping. In de Bakker, J. W., de Roever, W. P., and Rozenberg, G., 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.

[BHJ+87] Black, A. P., Hutchinson, N., Jul, E., Levy, H. M., and Carter, L. Distribution and abstract types in Emerald. *IEEE TSE*, 13(1):65–76, January 1987.

[Car88] Cardelli, L. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[CDD+89] Carrington, D., Duke, D., Duke, R., King, P., Rose, G., and Smith, P. Object-Z: An object oriented extension to Z. In *FORTE89, International Conference on Formal Description Techniques*, December 1989.

[CL91] Cusack, E. and Lai, M. Object-oriented specification in LOTOS and Z, or my cat really is object-oriented! In de Bakker, J. W., de Roever, W. P., and Rozenberg, G., editors, *Foundations of Object Oriented Languages*, pages 179–202. Springer Verlag, June 1991. LNCS 489.

[Cus91] Cusack, E. Inheritance in object oriented Z. In *Proceedings of ECOOP '91*. Springer-Verlag, 1991.

[DD90] Duke, D. and Duke, R. A history model for classes in object-Z. In *Proceedings of VDM '90: VDM and Z*. Springer-Verlag, 1990.

[Dha92] Dhara, K. K. Subtyping among mutable types in object-oriented programming languages. Master's thesis, Iowa State University, Ames, Iowa, 1992. Master's Thesis.

[DL92] Dhara, K. K. and Leavens, G. T. Subtyping for mutable types in object-oriented programming languages. Technical Report 92-36, Department of Computer Science, Iowa State University, Ames, Iowa, November 1992.

[DMN70] Dahl, O.-J., Myrhaug, B., and Nygaard, K. SIMULA common base language. Technical Report 22, Norwegian Computing Center, Oslo, Norway, 1970.

[GG89] Garland, S. and Guttag, J. 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.

[GHW85] Guttag, J. V., Horning, J. J., and Wing, J. M. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[GM87] Goguen, J. A. and Meseguer, J. Unifying functional, object-oriented and relational programming with logical semantics. In Shriver, B. and Wegner, P., editors, *Research Directions in Object Oriented Programming*. MIT Press, 1987.

[HM81] Hammer, M. and McLeod, D. A semantic database model. *ACM Trans. Database Systems*, 6(3):351–386, 1981.

[HO87] Halbert, D. C. and O'Brien, P. D. Using types and inheritance in object-oriented programming. *IEEE Software*, 4(5):71–79, September 1987.

[Hoa72] Hoare, C. Proof of correctness of data representations. *Acta Informatica*, 1(1):271–281, 1972.

[LD92] Leavens, G. T. and Dhara, K. K. 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.

[Lea89] Leavens, G. Verifying object-oriented prograsm that use subtypes. Technical Report 439, MIT Laboratory for Computer Science, February 1989. Ph.D. thesis.

[Lea91] Leavens, G. T. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[LG85] Liskov, B. and Guttag, J. *Abstraction and Specification in Program Design*. MIT Press, 1985.

[Lip92] Lipeck, U. Semantics and usage of defaults in specifications. In *Foundations of Information Systems Specification and Design*, March 1992. Dagstuhl Seminar 9212 Report 35.

[Lis92] Liskov, B. 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.

[LW90] Leavens, G. T. and Weihl, W. E. Reasoning about object-oriented programs that use subtypes. In *ECOOP/OOPSLA '90 Proceedings*, 1990.

[LW92] Liskov, B. and Wing, J. Family values: A semantic notion of subtyping. Technical Report 562, MIT Lab. for Computer Science, 1992. Also available as CMU-CS-92-220.

[LW94] Liskov, B. and Wing, J. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. and Systems*, pages 1811–1841, November 1994.

[Mey88] Meyer, B. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.

[MS90] Maier, D. and Stein, J. Development and implementation of an object-oriented DBMS. In Zdonik, S. and Maier, D., editors, *Readings in Object-Oriented Database Systems*, pages 167–185. Morgan Kaufmann, 1990.

[Nel91] Nelson, G. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[SCB+86] Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An introduction to Trellis/Owl. In *Proceedings of OOPSLA '86*, pages 9–16, September 1986.

[SH92] Scheid, J. and Holtsberg, S. Ina Jo specification language reference manual. Technical Report TM-6021/001/06, Paramax Systems Corporation, A Unisys Company, June 1992.

[Str86] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1986.

[ZW97] Zaremski, A. and Wing, J. Specification matching of software components. *ACM Trans. on Software Engineering and Methodology*, 6(4):333–369, October 1997.