Family Values: A Semantic Notion of Subtyping

Barbara Liskov and Jeannette M. Wing[1]
Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
USA

Abstract

The use of hierarchy is an important component of object-oriented design. Hierarchy allows the use of type families, in which higher level supertypes capture the behavior that all of their subtypes have in common. For this methodology to be effective, it is necessary to have a clear understanding of how subtypes and supertypes are related. This paper 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 paper also discusses the ramifications of this notion on the design of type families and on the contents of type specifications and presents a notation for specifying types formally.

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

---

[1]Permanent address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3890, wing@cs.cmu.edu, (412) 268-3068 (phone), (412) 681-5739 (fax).

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.

1. Introduction

What does it mean for one type to be a subtype of another? We argue that this is a semantic question having to do with the relationship between the behaviors of the two types. In this paper we give two ways to define the subtype relation; each definition relates specifications that describe the behavior of types. Our approach extends earlier work by providing for subtypes that have more methods than their supertypes, and by allowing sharing of mutable objects among multiple users. We discuss the ramifications of our approach with respect to various kinds of subtype relationships and give examples of type families that satisfy our definitions. We also present a formal language for specifying types; formal specifications allow us to give rigorous proofs of subtype relations.

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

    x: T := E

is considered to be legal provided the type of expression E is a subtype of the declared type T of variable x. Once the assignment has occurred, x will be used according to its ‘‘apparent’’ type T, with the expectation that if the program performs correctly when the actual type of x’s object is T, it will also work correctly if the actual type of the object denoted by x is a subtype of T. (In object oriented languages, classes are used to enable assignments like these and also as a code sharing mechanism. This paper discusses only the former topic.)

Clearly subtypes must provide the expected methods with compatible signatures. This consideration has led to the formulation by Cardelli of the contra/covariance rules [5]. However, these contra/covariance rules are not strong enough to ensure that the program containing the above assignment will work correctly for any subtype of T, since all they do is ensure that no type errors will occur. It is well known that type checking, while very useful, captures only a small part of what it means for a program to be correct; the same is true for the contra/covariance rules.

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

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

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

We present two definitions of the subtype relation, one using ''extension maps'' and the other using ''constraints.'' Either definition guarantees that all the invariant and history properties that hold for objects of the supertype also hold for objects of the subtype. In addition, either definition lets programmers reason directly in terms of specifications rather than the underlying mathematical models of types, be they algebras, categories, or higher-order lambda expressions. Our work is motivated by pragmatic concerns: we want to make our ideas accessible to everyday programmers. We provide a simple checklist that can be used by programmers in a straightforward way to validate a proposed design of a type hierarchy.

In addition to the definitions of the subtype relation, we also present a notation for giving formal specifications of types and subtypes. Our notation is based on the Larch specification language. It enables us to give rigorous proofs of subtype relations; example proofs are given in the appendices.

This paper makes three important technical contributions:

1. It provides two very general yet easy to use definitions of the subtype relation. Our definitions extend earlier work, including the most closely related work done by America [3], by allowing subtypes with mutable objects to have more methods than their supertypes.
2. It discusses the ramifications of the subtype relation and shows how interesting type families can be defined. For example, arrays are not a subtype of sequences (because the user of a sequence expects it not to change over time) and 32-bit integers are not a subtype of 64-bit integers (because a user of 64-bit integers would expect certain methods to succeed that will fail when applied to 32-bit integers). We show in Section 4 how useful type hierarchies that have the desired characteristics can be defined.
3. It presents a formal specification language and shows how rigorous proofs can be given. This latter work is important because it shows that the informal proofs that we expect from programmers (and present in the body of the paper) have a sound mathematical basis.

The paper is organized as follows. We describe our model of computation in Section 2. In Section 3 we present and discuss our first definition of subtyping, motivating it informally with an example relating stacks to bags. Section 4 discusses the ramifications of our definition on designing type hierarchies. In Section 5 we describe an alternative definition of the subtype relation that is motivated by exploring more carefully what goes into a type specification. Section 6 presents a technique for formally specifying types following the Larch approach. We describe related work in Section 7, and then close with a summary of contributions and open research problems. Details of the Larch specification language are described in Appendix I; the formal proofs are given in Appendix II.

2. Model of Computation

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

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

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

Given an object, x, and a state ρ with an environment, e, and store, s, we use the notation x_ρ to denote the value of x in state ρ; i.e., $x_\rho = \rho.s(\rho.e(x))$. When we refer to the domain of a state, dom(ρ), 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 legal values, and M is a set of methods. Each method for an object is a *constructor*, an *observer*, or a *mutator*. Constructors of an object of type τ return new objects of type τ; observers return results of other types; mutators modify the values of objects of type τ. A type is *mutable* if any of its methods is a mutator. 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, ρ_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[2]: 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:

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

Computations take place within a universe of shared, possibly persistent objects. Sharing can occur not only within a single program through aliasing, but also through multiple users accessing the same object through their separate programs. We assume the use of the usual mechanisms, e.g., locking, for synchronizing concurrent access to objects; we require that the environment uses these mechanisms to ensure the atomicity of the execution of each method invocation. We are also interested in persistence because we imagine scenarios in which a user might create and manipulate a set of objects today and

[2]This model is based on CLU semantics.

store them away in a persistent repository for future use, either by that user or some other user. In terms of database jargon, we are interested in concurrent transactions, where we are ignoring aborts and the need for recovery. The focus of this paper is on subtyping, not concurrency or recoverability; specific solutions to those problems should apply in our context as well.

3. The Meaning of Subtype

3.1. The Basic Idea

To motivate the basic idea behind our notion of subtyping, let's look at a simple-minded, slightly contrived example. Consider a bounded bag type that provides *put* and *get* methods that insert and delete elements into a bag. *Put* has a pre-condition that checks to see that adding an element will not grow the bag beyond its bound. *Get* has a pre-condition that checks to see that the bag is non-empty. Informal specifications [18] for *put* and *get* for a bag object, b, are as follows:

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

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

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

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

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

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

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

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

Suppose we want to show stack is a subtype of bag. We need to relate the values of stacks to those of bags. This can be done by means of an *abstraction function*, like that used for proving the correctness of implementations [10]. A given stack value maps to a bag value where we abstract from the insertion order on the elements.

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

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

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

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

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

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

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

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

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

3.2. Formal Definition

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

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

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

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

Figure 3.2 gives the definition of the subtype relation, <. The last clause of the definition requires what is shown in the diamond diagram in Figure 3-2, read from top to bottom. 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.

3.3. Discussion

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

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

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

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

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

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

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

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

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

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

Figure 3-1: Definition of the Subtype Relation

[3] We intentionally leave unspecified the language in which one writes a program, but imagine that it has the usual control structures, assignment, procedure call, etc.

Figure 3-2: The Diamond Diagram

America uses:

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

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

These two rules are the intuitive counterparts to the contravariant and covariant rules for signatures. The post-condition rule alone says that the subtype method's post-condition can be stronger than the supertype method's post-condition; hence, any property that can be proved based on the supertype method's post-condition also follows from the subtype's method's post-condition. Our pre-and-predicate rule differs from America's pre-and-post rule only when the subtype's method's pre-condition is satisfied and the supertype's method's pre-condition is not. In this case we do not require that the post-condition of the subtype's method imply that of the supertype's method; this makes sense because specifications do not constrain what happens when a pre-condition is not satisfied. Our weaker formulation gives more freedom to the subtype's designer. (A similar formulation is used by Leavens [15].)

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

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

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

The diamond rule is stronger than necessary because it requires equality between abstract values. We need only the weaker notion of *observable equivalence* (e.g., see Kapur’s definition [12]), since values that are distinct may not be observably different if the supertype’s set of methods (in particular, observers) is too weak to let us perceive the difference. In practice, such types are rare and therefore we did not bother to provide the weaker definition.

3.4. Applying the Definition of Subtyping as a Checklist

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

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

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

where the helping function, mk_elems: Sequence **of** Int $\rightarrow$ Multiset **of** Int, maps sequences to multisets. It is defined such that for all sq: Sequence **of** Int, i: Int:

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

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

Second, we define the renaming map, R:

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

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

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

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

or more formally[4],

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

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

1. A maps the 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.

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

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

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

To show this, we note first that since the two pre-conditions are equivalent, we can ignore them and deal with the post-conditions directly. (Thus we are proving America's stronger post-condition rule in this

[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.

case.) Next, we 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 directly from the definition of the abstraction function.

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

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

Showing the extension rule is just like showing that an implementation of a procedure satisfies the procedure's specification, except that we do not require equal values at the end, but just values that map via A to the same abstract value. (In fact, such a proof is identical to a proof showing that an implementation of an operation of an abstract data type satisfies its specification [10].) In doing the reasoning we rely on the specifications of the methods used in the program. Here is an informal argument for *swap_top*. We note first that since s.swap_top(i) terminates normally, so does the call on s.pop() (their pre-conditions are the same). *Pop* removes the top element, reducing the size of the stack so that *push*'s pre-condition holds, and then *push* puts i on the top of the stack. The result is that the top element has been replaced by i. Thus, $s_{\rho_2} = s_{\psi}$, where ρ_2 is the termination state if we run *swap_top* and ψ is the termination state if we run π. Therefore $A(s_{\rho_2}) = A(s_{\psi})$, since A is a function.

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

4. Type Hierarchies

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

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

4.1. Extension Subtypes

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

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

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

$$\forall\ s\text{: fat\_set}, \rho, \psi\text{: State} .\ [\ \rho < \psi \wedge s \in \text{dom}(\rho)\ ] \Rightarrow [\ \forall\ x\text{: int} .\ x \in s_\rho \Rightarrow x \in s_\psi\ ]$$

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

As a simple example of a many-to-one case, consider immutable pairs and triples. Pairs have methods that fetch the first and second elements; triples have these methods plus an additional one to fetch the third element. Triple is a subtype of pair and so is semi-mutable triple with methods to fetch the first, second, and third elements and to replace the third element. Here, E(x.replace(e)) = ε because the modification is not visible to users of the supertype. This example shows that it is possible to have a mutable subtype of an immutable supertype, provided the mutations are invisible to users of the supertype.

Figure 4-1: Pairs and Triples

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

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

Figure 4-2: Person, Student, and Employee

Another example from the database literature concerns different kinds of ships [24]. The supertype is ordinary ships with methods to determine such things as who is the captain and where the ship is registered. Subtypes contain more specialized ships such as tankers and freighters. There can be quite an elaborate hierarchy (e.g., tankers are a special kind of freighter). Windows are another well-known example [9]; subtypes include bordered windows, colored windows, and scrollable windows.

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

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

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

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

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

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

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

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

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

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

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

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

4.2. Constrained Subtypes

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

A very simple example concerns elephants. Elephants come in many colors (realistically grey and white, but we will also allow blue ones). However all albino elephants are white and all royal elephants are blue. Figure 4-3 shows the elephant hierarchy. The set of legal values for regular elephants includes all elephants whose color is grey or blue or white. The set of legal values for royal elephants is a subset of those for regular elephants and hence the abstraction function is into. The situation for albino elephants is similar.

Figure 4-3: Elephant Hierarchy

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

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

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

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

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

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

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

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

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

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

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

This simple example has led others to define a subtyping relation that requires non-monotonic reasoning [17], but we believe it is better to use a nondeterministic specification and straightforward reasoning methods. However, the example shows that a specifier of a type family has to anticipate subtypes and capture the variation among them in a nondeterministic specification of the supertype.

Another similar example concerns geometric figures. At the top of the hierarchy is the polygon type; it allows an arbitrary number of sides and angles of arbitrary sizes. Subtypes place various restrictions on these quantities. A portion of the hierarchy is shown in Figure 4-4.

Figure 4-4: Polygon Hierarchy

The bag type informally discussed in Section 3.1 is nondeterministic in two ways. As discussed earlier, the specification of *get* is nondeterministic because it does not constrain which element of the bag is removed. This nondeterminism allows stack to be a subtype of bag: The specification of *pop* constrains the nondeterminism. We could also define a queue that is a subtype of bag; its *dequeue* method would also constrain the nondeterminism of *get* but in a different way than *pop* does.

In addition, since the actual value of the bound for bags was not constrained, it can be any natural number, thus allowing subtypes to have different bounds. This nondeterminism shows up in the specification of *put*, where we do not say what specific bound value causes the call to fail. Therefore, a user of *put* must be prepared for a failure unless it is possible to deduce from past evidence, using the history property that the bound of a bag does not change, that the call will succeed. A subtype of bag might constrain the bound to a fixed value (as in the smallbag type), or to a smaller range. Several subtypes of bag are shown in Figure 4-5; largebags are essentially unbounded bags since their bound (fixed at creation) is ∞, and mediumbags have various bounds, so that this type might have its own subtypes, e.g., bag_150, containing all bags with bound equal to 150.

Figure 4-5: A Type Family for Bags

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

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

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

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

If we wanted a type family that included both dynamic_bag and bag, we would need to define a supertype in which the bound is allowed, but not required, to vary. Figure 4-6 shows the new type hierarchy where the *change_bound* method for varying_bag looks like:

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

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

Figure 4-6: Another Type Family for Bags

In the case of the bag family illustrated in Figure 4-5, all types in the hierarchy might actually be implemented. However, sometimes the supertypes are not intended to be implemented. These *virtual* *types* serve as placeholders for specific subtypes that are intended to be implemented; they let us define the properties all the subtypes have in common. Varying_bag is an example of such a type.

Virtual types are also needed when we construct a hierarchy for integers. Smaller integers cannot be a subtype of larger integers because of observable differences in behavior; for example, an overflow exception that would occur when adding two 32-bit integers would not occur if they were 64-bit integers. However, we clearly would like integers of different sizes to be related. This is accomplished by designing a nondeterministic, virtual supertype that includes them. Such a hierarchy is shown in Figure 4-7, 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 4-7: Integer Family

5. An Alternative Definition of the Subtype Relation

The definition of the subtype relation given in Section 3.2 relies on the existence of type specifications. In this section we discuss the content of specifications in more detail. This discussion leads us to another way of defining the subtype relation.

5.1. Contents of Specifications

A conventional specification for an abstract data type, τ, contains the following information:

- A *value space* that includes all the values contained by objects of type τ.
- A description of the behavior of each operation, including creators (recall that these are operations that do not take objects of type τ as arguments but return them as results).

Our specifications differ from conventional ones in two ways: (1) We define exactly the set of *legal* values; this might be the entire value space, but often it is a subset of the value space. (2) We describe the behavior of the objects' methods but we do not describe the behavior of the creators.

We do not include creators in specifications to avoid overconstraining subtypes. Subtype objects need to behave like supertype objects, but they need not come into existence in the same way. For example, when an elephant is created we would need to specify its color; however, when a royal elephant is created, no color need be indicated since all royal elephants are blue. Another example concerns pairs and tuples: the creator for a pair might take as arguments the values of the two components, while the creator for the triple would take as arguments the values of the triple's three components.

In addition, not including creators in specifications allows different implementations of a type to have different creators. For example one bag implementation might set the bound of a newly created bag to 100, while another might have the caller of the creator provide the bound.

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

To compensate for the lack of data type induction, we state the invariant explicitly in the type specification. For example, bag values are pairs, <elems, bound>, where elems is a multiset of integers. However, only pairs where the size of the elems multiset does not exceed the bound are legal bag values. This latter condition is the invariant.

This approach has three significant consequences. First, we need to prove that a type specification is *invariant-preserving*: Each method must preserve the type's invariant. (Also, creators must establish the invariant, as discussed in Section 6.2.) This property means that methods deal only with legal values of an object's type. To prove it, we assume each method is called on an object of type τ with a legal value (one that satisfies the invariant, I_τ) and show that any value of an object it produces or modifies is legal:

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

For example, we would need to show *put* and *get* each preserves the invariant for the bag type stated above. Informally the invariant holds because *put*'s pre-condition checks that there is enough room in the bag for another element and *get* decreases the size of the bag. (Appendix II.1 contains a proof of the invariant using notation developed in Section 6).

Second, we need to prove that abstraction functions *respect the invariant*. In the definition given in Section 3.2, S and T stand for sets of legal values. It is more convenient, however, to have them stand for value spaces. The abstraction function, A, is then defined over a value space, and we place additional checks on it to make sure it maps legal values to legal values. Thus we replace the first clause of our definition with:

1. The abstraction function, $A: S \rightarrow T$, must satisfy the following rule:
   - *Invariant rule*.

     $\forall$ s: S. $I_\sigma(s) \Rightarrow I_\tau(A(s))$

     where I_σ is the invariant for σ and I_τ is the invariant for τ.

This rule implies that A must be defined for all values of S that satisfy I_σ; A can be partial since it need not be defined for values of S that do not satisfy I_σ. Note that the requirement that the abstraction function preserves invariants, together with the methods and extensions rules, ensures that invariant properties of supertype objects hold for subtype objects.

The third consequence is that the absence of data type induction limits the kinds of invariant properties we can prove about objects. All invariant properties must follow from the conjunction of the type's invariant and invariants that hold for the entire value space. (For example, the size of all bag objects is greater than or equal to 0 because the size of the bag is equal to the size of the multiset component, and the multiset's size is greater than or equal to 0 because this is true for all multisets.) Since the explicit invariant limits what invariant properties can be proved, the specifier needs to be careful when defining it. The invariant must be strong enough to cut out values not of interest, but weak enough to allow definitions of interesting subtypes. For example, recall in the case of elephants that it would be an error to state that the color of an elephant is grey since this would rule out royal elephants as a subtype.

In summary, the invariant plays a crucial role in our specifications. It captures what normally would be proved through data type induction, allowing us to reason about properties of legal values of a type without having specifications of creators.

5.2. Alternative Definition

As stated in the introduction, we are interested in history properties as well as in invariants. History properties are predicates over sequences of states; they are especially of interest for mutable types.

We can formulate history properties as predicates over state pairs: for any computation, c

$$\forall\, x: \tau,\ \rho,\ \psi: \text{State}\ .\ [\ \rho < \psi \wedge x \in \text{dom}(\rho)\ ] \Rightarrow \phi(x_\rho, x_\psi)$$

where $\rho < \psi$ means that state ρ precedes state ψ in c. For example, in Section 4.1 we cast the property that elements in fat_sets never disappear in the above form. Notice that we implicitly quantify over all computations, c, and we do not require that ψ is the immediate successor of ρ.

Unlike type invariants, we need not make history properties explicit in a type specification. Instead they can be derived from a type's specification by showing that the property holds after the invocation of each of the type's methods. We actually need to do this only for each mutator:

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

where ϕ is a history property on objects of type τ.

However, it seems asymmetric to treat invariants and history properties differently. This leads us to consider what would happen if we include an explicit history property, which we will call a *constraint*, in a type specification. For example, we could add the following to the type specification of bag:

constraint b_ρ.bound = b_ψ.bound

to declare that a bag's bound never changes. The constraint must be proved to hold across all methods (by using the history rule); when this is true we say the specification *satisfies* the constraint. The constraint replaces the history rule as far users are concerned: users can make deductions based on the constraint but they cannot reason using the history rule directly. (The use of the term ''constraint'' is borrowed from the Ina Jo specification language [11], which also includes constraints in specifications.)

Explicit constraints allow us to simplify the definition of subtyping. Instead of the extension map, we just need to show that the subtype's constraint implies the supertype's (under the appropriate interpretation of subtype values using the abstraction function). The reason for the extension map is to ensure that any history property proved (through the history rule) for the supertype also holds for the subtype. The preservation of history properties is guaranteed for the non-extra methods (because of the methods rule). However, because the properties are not stated explicitly, we cannot prove them for the extra methods. Instead we need to ensure the extra methods satisfy any possible property, and this is surely guaranteed if the extra methods can be explained in terms of the non-extra methods. Showing that the subtype constraint is stronger than the supertype's takes care of all the methods, not just the extra ones.

The new definition of the subtype relation, which summarizes our explicit handling of invariants and constraints, is given in Figure 5-1. The first and third clauses replace those of our first definition in Figure 3.2. In this definition S and T stand for value spaces, I_σ is the invariant for σ, and I_τ is the invariant for τ. We assume each type specification preserves invariants and satisfies constraints.

As an example of how to use the new definition, consider a fat_bag type whose *get* method does not remove the returned element and a fat_stack type whose *pop* method does not remove the top element.

<u>Definition of the subtype relation, $<$</u>: $\sigma = \langle O_\sigma, S, M\rangle$ 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. 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_τ of τ is the corresponding renamed method m_σ of σ, the following rules must hold:
   - *Signature rule.*
     - *Contravariance of arguments.* m_τ and m_σ have the same number of arguments. If the list of argument types of m_τ is α_i and that of m_σ is β_i, then $\forall\ i.\ \alpha_i < \beta_i$.
     - *Covariance of result.* Either both m_τ and m_σ have a result or neither has. If there is a result, let m_τ's result type be γ and m_σ's be δ. Then $\delta < \gamma$.
     - *Exception rule.* The exceptions signaled by m_σ are contained in the set of exceptions signaled by m_τ.
   - *Methods rule.* For all $x: \sigma$:
     - *Pre-condition rule.* $m_\tau.pre[A(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre$.
     - *Predicate rule.* $m_\sigma.pred \Rightarrow m_\tau.pred[A(x_{pre})/x_{pre}, A(x_{post})/x_{post}]$
3. Subtype constraints ensure supertype constraints.
   - *Constraint Rule*. For all $x: \sigma$. $C_\sigma(x) \Rightarrow C_\tau[A(x_{pre})/x_\rho, A(x_{post})/x_\psi]$

Figure 5-1: An Alternative Definition of the Subtype Relation

Fat_bags never get smaller:

constraint $size(b_\rho) \leq size(b_\psi)$

and neither do fat_stacks:

constraint $length(s_\rho) \leq length(b_\psi)$

Intuitively, fat_stack preserves fat_set's constraint since the length of the sequence component of a stack is the same as the size of the multiset component of its bag counterpart. Notice that we do not have to say anything specific for *swap_top*.

Explicit constraints have pros and cons. On the plus side, they allow us to state the common properties of type families directly. For example, the varying_bag type discussed in Section 4.2 had both bag and dynamic_bag as subtypes. We allowed for these subtypes by defining an explicitly non-deterministic varying_bag method, *change_bound*. Both bag and dynamic_bag had this method; for bag it did not change the bound, while for dynamic_bag it did. With explicit constraints, we need not introduce this method (and therefore bag need not have it). Instead, varying_bag has just the *put* and *get* methods and the trivial constraint, ''true.'' Even though neither of these methods modifies the bound, a user of varying_bag is not entitled to assume that the bound never changes, because this fact is not stated in the constraint for varying_bag.

Another advantage is that 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.

On the minus side is the loss of the history rule. Although in our experience the desired constraint is usually obvious, nevertheless the specifier must be careful to define a strong enough constraint. For example, using the history rule we could deduce that once an element is added to a fat_bag it will always be there. However, with just the explicit constraint given above, users could not depend on this property (since they are not allowed to use the history rule). Of course, they ought not to depend on this property if we intend to allow fat_stack to be a subtype of fat_bag!

Another problem with explicit constraints is that people may expect to reason using the history rule. Therefore the consequences of a constraint must be made clear. For example, rather than stating a constraint of ‘‘true’’ for varying_bag, it would be better to say explicitly that both the bound and the elements may change with time:

$$\textbf{constraint}\ [\, b_\rho.\text{bound} = b_\psi.\text{bound} \vee b_\rho.\text{bound} \neq b_\psi.\text{bound} \,] \wedge \\ [\, \forall\, x\text{: Int} \,.\, x \in b_\rho.\text{elems} \Rightarrow (x \in b_\psi.\text{elems} \vee x \notin b_\psi.\text{elems}) \,]$$

In summary, having an explicit constraint is appealing because it allows us to state properties of type families declaratively, the subtype relation is simple, 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 that users can make use of it.

6. Formal Specifications

So far we have used informal specifications and our reasoning has therefore also been informal. To make our arguments more rigorous we need formal specifications. In this section we present a language to write formal specifications; we adopt the Larch specification language [13] for this purpose. This language enables us to prove subtype relationships within a formal proof system. For example, we formalize in Appendix II.3 the informal reasoning used in applying the checklist (Section 3.4) to show that stack is a subtype of bag.

In the sections below, we show how to specify types, creators, and subtypes. Our specifications contain explicit constraints, so that we are using the alternative definition of the subtype relation given in Figure 5-1. However, it is not difficult to switch to using the first definition of the subtype relation; we discuss this further in Section 6.3.

6.1. Type Specifications

Larch has two kinds of specifications. The first defines *sorts*, which are used to define the value spaces of types. Sorts are defined in a kind of specification called a *trait*; in general a trait will define one or more sorts. Several traits, which we use in our examples, are defined in the Appendices. For example, trait BBag defines a sort called B, whose values are similar to the pairs that we used for the value space of bags in our informal specification. More precisely, values in a sort are denoted by *terms*; we can use these terms to denote values of objects. Examples of terms in sort B are [{ }, 10] (the empty multiset with a bound of 10) and insert(3, [{ }, 10]) (the multiset resulting from inserting 3 into the empty multiset with bound 10). This latter term is equivalent to the simpler term [{3}, 10]; the BBag trait defines axioms that allow such equivalences to be proved. (In Larch, [ ] is used for tuples, and < > is used for sequences. We will be using Larch notation for the remainder of the paper.)

The second kind of specification is called an *interface*; it is used to specify types and procedures. Figure 6-1 shows the interface that specifies the bag type. The **uses** clause identifies the value space for bag as the sort B defined in the BBag trait. The remainder of the specification defines the invariant and constraint for the type, and the type's methods. The invariant and constraint are given in abbreviated form. The invariant is short for

$$\forall\ \rho\text{: State . } b \in \text{dom}(\rho) \Rightarrow \text{size}(b_\rho) \leq b_\rho\text{.bound}$$

and the constraint for

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

where ρ and ψ are states in some computation.

The form of the method specification is the same as that used earlier in our informal specifications; the only (but important) difference is that now the predicates appearing in the clauses have a rigorous meaning. The pre-condition is given in the **requires** clause; if this clause is missing, the pre-condition is bag = **type**

uses BBag (bag **for** B)

$\forall$ b: bag

invariant size(b_ρ) $\leq$ b_ρ.bound

constraint b_ρ.bound = b_ψ.bound

methods

put = **proc** (i: int)
requires size(b_{pre}) < b_{pre}.bound
modifies b
ensures b_{post} = insert(i, b_{pre})

get = **proc** () **returns** (int)
requires $\neg$isempty(b_{pre})
modifies b
ensures b_{post} = delete(result, b_{pre}) $\wedge$ result $\in$ b_{pre}

card = **proc** () **returns** (int)
ensures result = size(b_{pre})

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

end bag

Figure 6-1: A Type Specification for Bags

trivially ''true.'' The post-condition is the conjunction of the **modifies** and **ensures** clauses. In the **requires** and **ensures** clauses x stands for an object, x_{pre} stands for its value in the initial state, and x_{post} stands for the object's value in the final state.[5] 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.

A **modifies** $x_1, ..., x_n$ clause is shorthand for the following predicate

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

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

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

may change. Note that because the **modifies** clause is actually part of the post-condition, the predicate rule in our definition of subtyping also implies that a subtype's method cannot modify any objects that could not have been modified by the corresponding supertype's method.

Methods may terminate either normally or exceptionally. Specifiers can introduce values for exceptions that the method may signal through 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 isempty(b_{pre}) **then signal** empty
    else b_{post} = delete(result, b_{pre}) $\wedge$ result $\in$ b_{pre}

(Note that this specification of *get* is total whereas the one given in Figure 6-1 is partial.) Each method has a special implicit result parameter, *terminates*, whose value ranges over the exceptions listed plus the special value, normal, to stand for normal termination. The special predicate **signal** E stands for the assertion *terminates* = E; we assume normal termination (*terminates* = normal) unless otherwise explicitly specified.

In general, for a given method specification

m = **proc** (args) **returns** (result) **signals** (e1, ... en)
  requires ReqPred
  modifies x_1, ..., x_n
  ensures EnsPred

m.pre is ReqPred, m.post is ModPred $\wedge$ EnsPred, where ModPred is the predicate defined earlier for the meaning of a **modifies** clause, and m.pred, the method's associated first-order predicate, is:

ReqPred $\Rightarrow$ (ModPred $\wedge$ EnsPred)

We require that each type specification preserve the invariant and the satisfy the constraint, as defined in Sections 5.1 and 5.2.

6.2. Specifying Creators

For the reasons discussed in Section 5.1, a type specification does not include creators. Instead these are defined in separate interfaces. Figure 6-2 contains two interfaces defining creators for the type bag. As an argument to the special predicate **new**, *result* stands for the object returned, not its value. 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). The special **new** predicate, with the same meaning, would also appear in the post-conditions of constructors.

bounded_bag = **creators for** bag

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

end bounded_bag

simple_bag = **creators for** bag

  create = **proc** ( ) **returns** (bag)
    ensures **new**(result) $\wedge$ $result_{post}$ = [{ }, 100]

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

end simple_bag

Figure 6-2: Creator Specifications for Bags

All creators for a type τ must establish τ's invariant, I_τ:

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

This is similar to the requirement that each method of τ preserve the invariant.

To create bags, we call creators, after which we can invoke other bag methods. For example, consider

```
x: bag := bounded_bag$create(100)
y: bag := simple_bag$create_single(5)
x.put(5)
```

Using the specifications of the type and the creators, we know that at the end of calling *put*, x and y have the same value (they have the same elements and the same bound), but are different bag objects.

6.3. Subtype Specifications

Our notion of subtyping is defined by relating two type specifications. When specifying a subtype, it is convenient to be able to assert that a subtype relationship holds. The properties that must hold for the relationship to be legal can be checked at that time. The syntactic checks (signature rules) can be done automatically by the specification compiler; the semantic checks would require a theorem prover.

To assert that a type is a subtype of some other type, we simply append a *subtype clause* to its specification. We allow multiple supertypes; there will be a separate **subtype** clause for each. An example is given in Figure 6-3. Here the sort, S, for stack values is defined as a pair of a sequence and a limit; it is defined formally in trait BStack in Appendix II.3. The subtype clause gives the renaming and abstraction maps. Note in the abstraction function the use of ‘‘S’’ as the name of a sort. Note also the definition, as in Section 3.4, of the helping function *mk_elems*, which maps sort Seq (these are unbounded LIFO sequences) to M (these are unbounded multisets); these two sorts are defined in the Bag trait (in Appendix I) and LIFOSeq trait (in Appendix II.3).

```
stack = type

    uses Bstack (stack for S)

    ∀ s: stack

        invariant length(s_ρ) ≤ s_ρ.limit

        constraint s_ρ.limit = s_ψ.limit

        methods

            push = proc (i: int)
                requires length(s_pre) < s_pre.limit
                modifies s
                ensures s_post = add(i, s_pre)

            pop = proc () returns (int)
                requires s_pre ≠ <>
                modifies s
                ensures result = top(s_pre) ∧ s_post = rem(s_pre)

            swap_top = proc (i: int)
                requires s_pre ≠ <>
                modifies s
                ensures s_post = add(i, rem(s_pre))

            height = proc () returns (int)
                ensures result = length(s_pre)

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

    subtype of bag  (push for put, pop for get, length for card)

        ∀ s: S . A(s) = [mk_elems(s.items), s.limit]
            where mk_elems: Seq → M
                ∀ i: Int, sq: Seq
                    mk_elems(<>) = { }
                    mk_elems(add(i, sq)) = insert(i, mk_elems(sq))

end stack
```

Figure 6-3: Stack Type

Although in general a subtype's trait is different from any of its supertypes, we often use the same trait to define an entire family of types. For example, a smallbag type would probably be defined using BBag. What distinguishes the smallbag type from its supertype is its more constraining invariant. Not only must the size of a smallbag not exceed its bound but also its bound must be equal to 20:

invariant $size(s_\rho) \leq s_\rho.bound \wedge s_\rho.bound = 20$

Finally, notice why we do not consider invariants as shorthand for explicit conjuncts in a method's pre- and post-conditions. If they were written explicitly as part of the pre- and post-condition then the pre-condition rule would require in general that the supertype's invariant implies a subtype's. Usually just the opposite holds. For example, 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.

We have defined types and subtypes using explicit constraints. However, we can easily adapt our notation to using extension maps instead. In this case, there would not be any **constraint** part of the specification, and each subtype clause would define the extension map E in addition to defining A and R. The only problem is that to do rigorous proofs, we would need to have a formally defined language in which to write the explanations. Defining such a language is not difficult, but we will not go into the details in this paper. Appendix II.3 contains proofs for both the constraint rule and the extension rule.

7. Related Work

Research on defining subtype relations can be divided into two categories: work on the ''syntactic'' notion of subtyping and work on the ''semantic'' notion. We clearly differ from the syntactic notion, formally captured by Cardelli's contra/covariance rules [5] and used in languages like Trellis/Owl [25], Eiffel [8], POOL [2], and to a limited extent Modula-3 [22]. Our rules place constraints not just on the signatures of an object's methods, but also on their semantic behavior as described in type specifications. Cardelli's rules are a strict subset of ours (ignoring higher-order functions).

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

Our work is most closely related to that of America [3] who uses the stronger pre- and post-condition rules as discussed in Section 3. (Meyer also uses these rules for Eiffel [8], although here the pre- and post-conditions are given ‘‘operationally,’’ by providing a program to check them, rather than assertionally.) However, America discusses only the meaning of the subtype methods that simulate those of the supertype, ignoring the problems introduced by the extra mutators.

Our work is also similar to America’s in its approach: We expect programmers to reason directly in terms of specifications; we call this approach ‘‘proof-theoretic.’’ Most other approaches are ‘‘model-theoretic’’; programmers are expected to reason in terms of mathematical structures like algebras or categories. We believe a proof-theoretic approach is better because it is much more accessible to programmers. We also go one step further than America by giving a specific formalism with which to do proofs from specifications.

The emphasis on semantics of abstract types is a prominent feature of the work by Leavens. We go further by addressing mutable abstract types. In his Ph.D. thesis [14] Leavens defines a model-theoretic semantic notion of subtyping. He defines types in terms of algebras and subtyping in terms of a *simulation relation* between them. Further work by Leavens and Weihl showing how to verify programs with subtypes uses Hoare-style reasoning as we do [16]. Again, their work is restricted to immutable types. Their simulation relations map supertype values down to subtype values; hence, they do reasoning in the subtype value space. In contrast we use our abstraction function to map values up to the supertype value space. We can rely on the substitutivity property of equality; they cannot. Indeed, in our proofs we depend on A being a function.

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

In his 1992 Master’s thesis [6], Dhara extends Leavens’ thesis work to deal with mutable types. Again, his approach is model-theoretic and based on simulation relations; moreover, because of a restriction on aliasing in his model, his definition disallows certain subtype relations from holding that we could allow. Dhara has no counterpart to our extension or constraint rule, and no techniques for proving subtype relations.

To our knowledge, Utting is the only other researcher to take a proof-theoretic approach to subtyping [26]. His formalism is cast in the refinement calculus language [23], an extension of Dijkstra's guarded command language [7]. Utting makes a big simplifying assumption: he does not allow data refinement between supertype and subtype value spaces. Our use of abstraction functions directly addresses this issue, which intuitively is the heart of any subtyping relation.

Finally, our extension rule is related to the more general work done on relating the behaviors of two different concurrent processes. Such work includes Milner's CCS [21] and Lynch's I/O automata [20]. Abadi and Lamport's *refinement mappings* [1] are akin to our extension mappings. To our knowledge, these models for reasoning about concurrent systems have not yet been applied in the context of subtyping.

In summary, our work is similar in spirit to America and Utting because they take a proof-theoretic approach to defining a semantic notion of subtyping. It complements the model-theoretic approach taken by Leavens, Leavens and Weihl, Bruce and Wegner, and Dhara. Only America, Utting, and Dhara deal with mutability, but none has formulated the essence of our extension or constraint rule.

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 the extension rule to reason about the extra methods, and the other using constraints. Either of these approaches guarantees that subtypes preserve their supertype's 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. A formal specification method enables us to define type semantics, i.e., behavioral properties about their values and methods; it also provides a framework within which to do formal reasoning about programs. It provides a sound basis for our informal specifications, subtyping checklists, and proofs of invariant and history properties.

One reason we chose Larch for our formalism is that it takes a ‘‘proof-theoretic’’ view towards defining semantics of specifications. This view means that formal proofs can 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 [19].

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.

I. A Larch Shared Language Refresher

Formalizations of the informal proofs given in the paper rely on properties of the value spaces of types. Since we use Larch traits to specify formally these value spaces, we begin here with a tutorial on traits. In the next Appendix, we give details of proofs of invariants, constraints, and subtype relationships.

Figure I-1 presents an example of a trait. The Mset trait is useful for defining the value space for unbounded multisets. It introduces *sorts*, e.g., M, that are used to distinguish terms (and hence the values they denote) much like types are used to distinguish objects. A trait also introduces *function* symbols, e.g., {} and insert, that provide a trait's term language. For example, the term {} denotes the empty multiset value and the term insert(a, {}) denotes the multiset with the single element a in it. Mset **includes** the traits, Integer and Natural, which used for defining integers of sort Int and natural numbers of sort N; they introduce functions like 0 and + with the obvious meanings.

We use equations, **generated by**, and **partitioned by** clauses to constrain the meaning of the trait functions. The equations in the trait define an equality relation on terms. Two equal terms denote the same value. The **generated by** clause introduces an inductive rule of inference that allows us to prove properties about all terms of sort M. For example, from this inductive rule, we could prove the invariant that the size of all multisets is always non-negative. The **partitioned by** clause introduces more equalities between terms; it says that two terms of sort M are the same if they cannot be distinguished by using the count function. Thus, two multisets are the same if the counts of their elements are the same. For example, we could prove that the two terms insert(insert(insert({}, a), b), a) and insert(insert(insert({}, a), a), b) denote the same multiset value (order of insertion is irrelevant) but that they are both different from insert(insert({}, a), b) (the number of times each element appears matters).

The equations, **generated by**, **partitioned by**, and the standard axioms and rules of inference for

```
MSet: trait

    includes Integer(Int), Natural(N)

    introduces
        {}:  -> M
        insert: Int, M -> M
        delete: Int, M -> M
        count: Int, M -> N
        __∈ __: Int, M -> Bool
        size: M -> N
        isempty: M -> Bool
        __∪__: M, M -> M

    asserts
        M generated by {}, insert
        M partitioned by count
        ∀ m, m1: M, i: i1, i2: Int
          count(i, {}) == 0;
          count(i1, insert(i2, m)) == count(i1, m) + (if i1 = i2 then 1 else 0);
          count(i1, delete(i2, m)) == count(i1, m) ⊖ (if i1 = i2 then 1 else 0);
          i1 ∈ {} == false;
          i1 ∈ insert(i2, m) == i1 = i2 ∨ i1 ∈ m;
          size({}) == 0;
          size(insert(i, m)) == size(m) + 1
          isempty({}) == true
          isempty(insert(i, m)) == false
          count(i, m ∪ m1) == count(i, m) + count(i, m1)
```

Figure I-1: Multiset Trait

first-order predicate logic with equality together define the first-order theory of a trait.

Figure I-2 presents another trait, the BBag trait, which is useful for defining the values of bounded bags. This trait **includes** Mset. In Larch, when a trait S **includes** another trait T, it is as if all equations, **generated by**, and **partitioned by** clauses of T are written in S explicitly.

Bounded bag values are tuples. The Larch **tuple of** construct is shorthand for introducing fixed-length tuples. Bounded bags of sort B are represented as a pair of a multiset, elems, and a bound, bound. B is **generated by** the Larch tuple constructor denoted by square brackets, i.e., [__, __]: M, Int -> B. Each field name defines two distinct functions for getting (__.elems, __.bound) and setting (set_elems, set_bound) the value for that field. B is **partitioned by** the retrieval functions, __.elems and __.bound. In the equations, we do not retrieve the parts of the tuple explicitly, but instead use the pattern matching notation [m, n].

The functions on bounded bag values are defined in terms of functions for the unbounded multiset

BBag: **trait**

includes Mset
B **tuple of** elems: M, bound: N

introduces
insert: Int, B -> B
delete: Int, B -> B
count: Int, B -> Int
$\_\_ \in \_\_$: Int, B -> Bool
size: B -> N
isempty: B -> Bool

asserts $\forall$ m: M, i: Int, n: N
insert(i, [m, n]) == [insert(i, m), n]
delete(i, [m, n]) == [delete(i, m), n]
count(i, [m, n]) == count(i, m)
i $\in$ [m, n] == i $\in$ m
size([m, n]) == size(m)
isempty([m, n]) == isempty(m)

Figure I-2: Bounded Bag Trait

values in the obvious manner. Notice there is no constraint on the relationship between the size of the elems component of the tuple and its bound. Such constraints are given when defining type specifications.

II. Details of Proofs

II.1. Showing the Type Invariant for Bag

To show that a type invariant holds, we must show that it is preserved by all methods of the type. We must also show that it is established by every creator of an object of that type.

The type invariant for bag is:

$size(b_\rho) \leq b_\rho.bound$

Lemma: $\forall$ bval: B . size(insert(i, bval)) = 1 + size(bval) By induction on b.elems.

Proof of inductive step:
For each method, we need to show that assuming $size(b_{pre}) \leq b_{pre}.bound$,
$size(b_{post}) \leq b_{post}.bound$.

Case 1: put

Show $size(b_{post}) \leq b_{post}.bound$
$size(insert(i, b_{pre})) \leq (insert(i, b_{pre})).bound$ Substituting for b_{post} from post-condition.
$1 + size(b_{pre}) \leq [insert(i, b_{pre}.elems), b_{pre}.bound].bound$ By Lemma.

$1 + size(b_{pre}) \leq b_{pre}.bound$
True since $size(b_{pre}) < b_{pre}.bound$ from put's pre-condition.

Case 2: get

Show $size(b_{post}) \leq b_{post}.bound$
$size(delete(result, b_{pre})) \leq (delete(result, b_{pre})).bound$
$size(delete(result, b_{pre})) \leq [delete(result, b_{pre}.elems), b_{pre}.bound].bound$
$size(b_{pre}) - 1 \leq b_{pre}.bound$ Def'n of size, delete, and get's pre-condition.
True since $size(b_{pre}) \leq b_{pre}.bound$ from assumption.

Cases 3 and 4: card and equal. Trivial since they both do not change b.

□

Proof of the basis step: Show $size(result_{post}) \leq result_{post}.bound$

We will show this for the creator of the bounded_bag interface specification in Figure 6-2. The others are similar.

$size([\{\}, n]) \leq [\{\}, n].bound$	Substituting in for $result_{post}$ from create's post-condition.
$0 \leq n$	Def'n of size, bound
$n \geq 0$	Create's pre-condition and arithmetic.

□

II.2. Showing a Constraint Property

In this section we show that the bound of a bag b never changes, i.e., that it satisfies this constraint:

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

For each bag mutator, we need to show:

$b_{pre}.bound = b_{post}.bound$

Proof:
Case 1: m = put
By put's post-condition
$b_{post} = insert(i, b_{pre})$

$b_{post}.bound = (insert(i, b_{pre})).bound$
$= [insert(i, b_{pre}.elems), b_{pre}.bound].bound$
$= b_{pre}.bound$

Case 2: m = get
By get's post-condition
$b_{post} = delete(result, b_{pre})$
$b_{post}.bound = (delete(result, b_{pre})).bound$
$= [delete(result, b_{pre}.elems), b_{pre}.bound].bound$
$= b_{pre}.bound$

□

II.3. Proving that Stack is a Subtype of Bag

Section 6 presented a precise way of writing down the specifications of types and subtypes, thereby giving us a way to show more rigorously that a particular subtype relationship holds. The formal proof for showing that stack is a subtype of bag shows more explicitly how the abstraction function lets us reason in terms of the values of bags and stacks using the vocabularies from their corresponding traits.

The type specification for stacks (Figure 6-3) uses the traits given in Figures II-1 and II-2. LIFOSeq defines a value space for unbounded LIFO sequences (it is like the traditional algebraic specification for stacks). Stack values are tuples containing a sequence of integers, items, and a bound, limit.

```
LIFOSeq: trait

    includes Integer, Natural (N)

    introduces
        <>: -> Seq
        add: Int, Seq -> Seq
        rem: Seq -> Seq
        top: Seq -> Int
        length: Seq -> N

    asserts
        Seq generated by <>, add
        Seq partitioned by rem, top, length
        ∀ i: Int, s: Seq
            top(add(i, s)) == i;
            rem(add(i, s)) == s;
            length({}) == 0;
            length(add(i, s)) == length(s) + 1
```

Figure II-1: Unbounded LIFO Sequence Trait

To prove that stack is a subtype of bag, we first assume that the type specifications are invariant-preserving. In Appendix II.1 we showed that the bag specification is invariant-preserving; the proof for the stack specification is similar.

Then, for either the original or alternative definition, we define (1) an abstraction function and (2) a renaming map and show that each obeys certain rules. (Below, we also show that the abstraction function as defined preserves invariants.) Finally, if using the original definition of subtyping, we define (3a) an extension map and show that it obeys the diamond property; if using the alternative, we show that (3b) the bag constraint is satisfied by stack's methods (assuming each type specification satisfies its constraint).

```
BStack: trait

    includes LIFOSeq
    S tuple of items: Seq, limit: N

    introduces
        add: Int, S -> S
        rem: S -> S
        top: S -> Int
        length: S -> N

    asserts
        ∀ i: Int, s: Seq, n: N
            add(i, [s, n]) == [add(i, s), n]
            top([s, n]) == top(s)
            rem([s, n]) == [rem(s), n]
            length([s, n]) == length(s)
```

Figure II-2: Stack Trait

1. The Abstraction Function

We use the following abstraction function:

```
A: S -> B
mk_elems: Seq -> M

For all s: S, sq: Seq, i: Int

    A(s) = [mk_elems(s.elems), s.bound]
    mk_elems(<>) = {}
    mk_elems(add(i, sq)) = insert(i, mk_elems(sq))
```

Before we show that any of the rules hold, let's prove some lemmas:

Lemma 1: ∀ s: Seq. size(mk_elems(s)) = length(s)

Proof by induction.

Base: s = <>

```
size(mk_elems<>) = length(<>)
size({})  = length(<>)
0  = 0
```

Ind.: Assume size(mk_elems(sq)) = length(sq). (IH)

Let s = add(i, sq)

size(mk_elems(add(i, sq))) = length(add(i, sq)	
size(insert(i, mk_elems(sq))) = length(add(i, sq))	Def'n of mk_elems.
size(mk_elems(sq)) + 1 = length(sq) + 1	Def'n of size and length for BBag and BStack.
true	By IH.

□

Lemma 2: ∀ s: S. size(A(s)) = length(s)

Let s = [sq, n]

size(A([sq, n])) = length(s)	
size([mk_elems(sq), n]) = length([sq, n])	Def'n of A.
size(mk_elems(sq)) = length(sq)	Def'n of size and length in BBag and BStack.
true	By Lemma 1.

□

Lemma 3: ∀ s: S. A(s).bound = s.limit

Let s = [sq, n]

A([sq, n]).bound = [sq, n].limit	
[mk_elems(sq), n].bound = [sq, n].limit	Def'n of A.
n = n	Def'n of .bound and .limit.

□

Abstraction Rules

We need to show A preserves invariants. By definition, we know that A is defined only for stacks that satisfy the stack type invariant:

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

We need to show that it defines bag values that satisfy the bag type invariant:

$size(b_\rho) \leq b_\rho.bound$

Proof:

Case 1: s = [<>, n] and A(s) = b = [{}, n]

size([{}, n]) ≤ [{}, n].bound
size({}) ≤ n
0 ≤ n

Case 2: s = [add(i, sq), n] and A(s) = b = [insert(i, mk_elems(sq)), n]
Since A is defined, we know that
length(add(i, sq)) ≤ n   By stack's invariant
length(sq) + 1 ≤ n
length(sq) ≤ n - 1

size([insert(i, mk_elems(sq)), n]) ≤ [insert(i, mk_elems(sq)), n].bound
size(insert(i, mk_elems(sq))) ≤ n
1 + size(mk_elems(sq)) ≤ n
1 + length(sq) ≤ n   By Lemma 1
True, since length(sq) ≤ n - 1

□

2. The Renaming Function

The renaming function, R, is trivially defined as follows:

$R: M_{stack} \rightarrow M_{bag}$
R(push) = put
R(pop) = get
R(height) = card
R(equal) = equal

<u>Methods Rule</u>

Let's look at the pre-condition and predicate rules for just one method, *push*.

Push's pre-condition:

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

Proof:

$length(s_{pre}) < s_{pre}.limit \Rightarrow length(s_{pre}) < s_{pre}.limit$ By Lemmas 2 and 3.

□

Before we show the predicate rule for *push*, let's prove another lemma:

Lemma 4: A(add(i, s)) = insert(i, A(s))

Proof:

A(add(i, s)) = insert(i, A(s))	
A([add(i, s.elems), s.bound]) = insert(i, A([s.elems, s.bound]))	Def'n of add.
[mk_elems(add(i, s.elems)), s.bound] = insert(i, [mk_elems(s.elems), s.bound])	Def'n of A.
[insert(i, mk_elems(s.elems)), s.bound] = [insert(i, mk_elems(s.elems)), s.bound]	Def'n of mk_elems.

□

Push's predicate rule:

$length(s_{pre}) < s_{pre}.limit \Rightarrow s_{post} = add(i, s_{pre}) \wedge$ **modifies** s
$\Rightarrow$
$size(A(s_{pre})) < A(s_{pre}.bound \Rightarrow A(s_{post}) = insert(i, A(s_{pre})) \wedge$ **modifies** s

Proof:

By Lemmas 2 and 3, it suffices to show that the subtype's post-condition implies the supertype's:

$s_{post} = add(i, s_{pre}) \wedge$ **modifies** s
$\Rightarrow$
$A(s_{post}) = insert(i, A(s_{pre})) \wedge$ **modifies** s

This reduces to showing:

$s_{post} = add(i, s_{pre}) \Rightarrow A(s_{post}) = insert(i, A(s_{pre}))$

Substituting add(i, s_{pre}) from LHS for s_{post} in RHS of the implication:

$A(add(i, s_{pre})) = insert(i, A(s_{pre}))$

which is true by Lemma 4.

□

3a. The Extension Map

The only method requiring a non-trivial explanation via the extension map is *swap_top*; its effect is the same as a *pop* followed by a *push*:

$E: O_{stack} \times M_{stack} \times Obj^* \rightarrow Prog$
E(s.swap_top(i)) = s.pop(); s.push(i)

<u>Diamond Rule</u>

Proof: We need to show

$\forall\ \rho_1, \rho_2, \psi_1, \psi_2$: State
ρ_1 s.swap_top(i) $\rho_2 \wedge \rho_1$ s.pop() ψ_1 s.push(i) ψ_2
$\Rightarrow A(s_{\rho_2}) = A(s_{\psi_2})$

1. Consider ρ_1 s.swap_top(i) ρ_2.

$s_{\rho_2} = add(i, rem(s_{\rho_1}))$ By swap_top's post-condition.

2. Consider ρ_1 s.pop() ψ_1 s.push(i) ψ_2.

$s_{\psi_1} = rem(s_{\rho_1})$ By pop's post-condition.
$s_{\psi_2} = add(i, s_{\psi_1})$ By push's post-condition.
$s_{\psi_2} = add(i, rem(s_{\rho_1}))$ By substitution.

3. $s_{\rho_2} = s_{\psi_2}$ From 1 and 2 and by def'n of equality.

4. $A(s_{\rho_2}) = A(s_{\psi_2})$ Since A is a function.

□

Notice that we use stack's definition of *push* and *pop* in the proof and that we rely on A's being a function.

3b. Preserves Constraints

Suppose we were to use the alternate definition where constraints are specified explicitly and we do not have an extension map. In Appendix II.2 we showed that the bag specification satisfies its constraint that its bound never changes. The constraint and proof of satisfaction for the stack type would be similar:

constraint s_ρ.limit = s_ψ.limit

To show stack is a subtype of bag, we need to show that stack's constraint implies bag's, under the abstraction function:

$s_{pre}.limit = s_{post}.limit \Rightarrow b_{pre}.bound = b_{post}.bound\ [A(s_{pre})/s_{pre}, A(s_{post})/s_{post}]$

This is trivial using Lemma 3.

A more interesting example is to show that fat_stack's constraint:

constraint length(s_ρ) $\leq$ length(s_ψ)

preserves fat_bag's:

constraint size(b_ρ) $\leq$ size(b_ψ)

This is trivial using Lemma 2.

References

[1] M. Abadi and L. Lamport.
The existence of refinement mappings.
Technical Report 29, Digital Equipment Corp./Systems Research Center, August, 1988.

[2] Pierre America.
A Parallel Object-Oriented Language with Inheritance and Subtyping.
SIGPLAN Notices 25(10):161-168, October, 1990.

[3] Pierre America.
Designing an Object-Oriented Programming Language with Behavioural Subtyping.
In J. W. de Bakker and W. P. de Roever and G. Rozenberg (editor), *LNCS*. Volume 489: *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, pages 60-90. Springer-Verlag, NY, 1991.

[4] K.B. Bruce and P. Wegner.
An Algebraic Model of Subtypes in Object-Oriented Languages (Draft).
In *ACM SIGPLAN Notices*. October, 1986.
Object-Oriented Programming Workshop.

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

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

[7] Edsger W. Dijkstra.
A Discipline of Programming.
Prentice Hall, 1976.

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

[9] Daniel C. Halbert and Patrick D. O'Brien.
Using Types and Inheritance in Object-Oriented Programming.
IEEE Software :71-79, September, 1987.

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

[11] 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.

[12] Deepak Kapur.
Towards a Theory of Abstract Data Types.
Technical Report 237, MIT LCS, June, 1980.
Ph.D. thesis.

[13] John V. Guttag, James J. Horning and Jeannette M. Wing.
The Larch Family of Specification Languages.
IEEE Software 2(5):24-36, sept, 1985.

[14] Gary Leavens.
Verifying Object-Oriented Prograsm That Use Subtypes.
Technical Report 439, MIT Lab. for Computer Science, February, 1989.
Ph.D. thesis.

[15] Gary T. Leavens and William E. Weihl.
Subtyping, Modular Specification, and Modular Verification for Applicative Object-Oriented Programs.
Forthcoming.

[16] Gary T. Leavens and William E. Weihl.
Reasoning about Object-Oriented Programs that use Subtypes.
In *ECOOP/OOPSLA '90 Proceedings*. 1990.

[17] Udo Lipeck.
Semantics and Usage of Defaults in Specifications.
In Hans-Dieter Ehrich and Joseph A Goguen and Amilcar Sernadas (editors), *Foundations of Information Systems Specification and Design*. March, 1992.
Dagstuhl Seminar 9212 Report 35.

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

[19] S.J. Garland and J.V. Guttag.
An Overview of {L
{P}, the {L}arch {P}rover.}
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.

[20] N. Lynch and M. Tuttle.
Hierarchical correctness proofs for distributed algorithms.
In *Proc. of 6th ACM Symposium on Principles of Distributed Computation*, pages 137-151. August, 1987.

[21] R. Milner.
Communication and Concurrency.
Prentice Hall, 1989.

[22] Greg Nelson, editor.
Systems Programming with Modula-3.
Prentic Hall, -1991.

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

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

[25] Craig Schaffert, Topher Cooper and Carrie Wilpolt.
Trellis: Object-Based Environment Language Reference Manual.
Technical Report 372, Digital Equipment Corp./Easter Research Lab., 1985.

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

Table of Contents

List of Figures