Specifications and Their Use in Defining Subtypes

Barbara Liskov
Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

Jeannette M. Wing
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Specifications are useful because they allow reasoning about objects without concern for their implementations. Type hierarchies are useful because they allow types that share common properties to be designed as a family. This paper is concerned with the interaction between specifications and type hierarchies. We present a way of specifying types, and show how some extra information, in addition to specifications of the objects' methods, is needed to support reasoning. We also provide a new way of showing that one type is a subtype of another. Our technique makes use of information in the types' specifications and works even in a very general computational environment in which possibly concurrent users share mutable objects.

1. Introduction

Object-oriented programming languages support a programming methodology based on data abstraction in which programs are composed of modules, each implementing an abstract data type. The type is abstract because it is possible to interact with its objects only by calling their operations or methods. The type's implementation (e.g., a class) defines a representation for the type's objects (e.g., a set of instance variables) and provides implementations of the methods based on this representation. The representation details are encapsulated: they are accessible only to the class, and hidden from users.

Encapsulation is useful because it allows us to reimplement a type with another class without affecting its users, assuming the new class has the same behavior as the old one. However, to make sense out of this ‘‘behavior’’ requirement we need a way of defining what the required behavior is. The old implementation is not a sufficient definition since it includes details that may or may not be important. For example, if a collection object has a method that returns an element that matches some predicate, the implementation of the method will make a choice if there are several elements that match. Is this choice part of the required behavior? Or, can a different implementation make a different choice?

The way to capture behavior is to define it separately from implementations in a *specification*. Specifications have long been a cornerstone of the data abstraction methodology, but have received less attention in work on object-oriented programming. Furthermore, specification techniques for data abstractions aren't quite right for object types because they assume methods belong to the type, not the object.

This paper provides a specification technique tailored to the needs of object types. Our approach allows a type to have multiple implementations and makes it convenient to define the subtyping relation. Our specifications are based on the Larch formal specification technique, which means that they have a precise mathematical meaning that serves as a firm foundation for reasoning, e.g., that a class implements a type correctly. We also discuss informal specifications based on our approach.

Specifications are used to reason about object behavior in abstract terms. Two kinds of properties are of interest: *invariant* properties, which are properties true of all states, and *history* properties, which are properties true of all sequences of states. For example, an invariant property of an integer counter might be that its value is always greater than or equal to zero; a history property might be that its value always increases. 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., the value of the integer counter eventually reaches 100, but our focus here will be just on safety properties.

Our type specifications define the behavior of methods of objects of the type (i.e., the ‘‘instance methods’’) but not any of the additional methods, usually called ‘‘class methods’’, that do not belong to particular objects. These additional methods are used to create new objects from scratch (we will refer to such methods as *creators*) and may also be used for additional purposes, e.g., to maintain statistics about various properties of the objects. Leaving these methods out is desirable because this is a place where implementations might differ (e.g., different classes that implement the same type might have different creators); in addition, different creators may be needed for subtypes.

In the absence of the creators, however, there is no way to prove invariant properties. The problem is that such properties are proved by *data type induction*, in which the creators are used for the basis step, and the methods for the induction step. If there are no creators defined in the specification, there is no basis step. To compensate for this loss, we add explicitly stated *invariants* to our specifications.

Specifications are also useful for defining type hierarchy. 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. Intuitively, 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 gives a definition of the subtype relation that ensures the subtypes’ objects behave properly. Our definition ensures that all history and invariant properties that can be proved about supertype objects also hold for subtype objects. In particular, just as we add invariants to specifications to state a type’s invariant properties, we add *constraints* to state its history properties explicitly. Proofs that a subtype ensures constraints of a supertype are done in terms of type specifications directly.

Thus, the paper makes two contributions:

1. It provides a way of specifying object types that allows a type to have multiple implementations and makes it convenient to define the subtyping relation. The technique requires creators to be specified separately from types, but still supports the invariant properties needed for reasoning about the type’s objects.

2. It provides a new definition of the subtype relation. Our technique requires that additional information be included in the type’s specification. It works even in a very general environment in which possibly concurrent users share mutable objects. Our technique is also constructive: One can prove whether a subtype relation holds by proving a small number of simple lemmas.

Others have worked on both of the problems attacked in this paper. For example, many have proposed Z as the basis of specifications of object types [7, 10, 5]; Goguen and Meseguer use FOOPS [13]; Leavens and his colleagues use Larch [16, 18, 9]. Though many of these researchers separate the specification of an object’s creators from its other methods, no one has identified the problem posed by the missing creators, and thus no one has provided an explicit solution to this problem.

Others such as America [1], Cusack [6], and Dhara and Leavens [9] have proposed rules for determining whether one type is a subtype of another. Many of these approaches are not constructive, i.e., they tell you what to look for, but not how to prove that you got it. Other work [27, 17] is couched in formalisms that we believe are not very easy for programmers to deal with. In contrast, our subtype definition is constructive and takes the form of a simple checklist of rules, which programmers can use in a formal or informal way. Furthermore, only we have a technique that works in a general environment in which objects can be shared among possibly concurrent users. The rule for proving the subtype relation given in this paper is simpler than the one given in our own earlier work [22], but it requires more information in specifications, which may be a disadvantage.

The remainder of the paper is organized as follows. In Section 2 we describe our model of computation. Section 3 describes our specification technique and the extra information needed to make up for the loss of data type induction. Section 4 describes our subtype relation and the extra information needed for it to work in a very general computational environment. We close with a summary of what we have accomplished.

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

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

$$\mathit{State} = \mathit{Env} \times \mathit{Store}$$
$$\mathit{Env} = \mathit{Var} \rightarrow \mathit{Obj}$$
$$\mathit{Store} = \mathit{Obj} \rightarrow \mathit{Val}$$

Given an object, x, and a state, ρ, with an environment, ρ.e, and store, ρ.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 $O \subseteq \mathit{Obj}$ is a set of objects, $V \subseteq \mathit{Val}$ is a set of values, and M is a set of methods. Each method for an object is a *constructor*, an *observer*, or a *mutator*. Constructors of an object of type 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[1]: the environment can change through assignment; the store can change through the invocation of a mutator; the domain can change through the invocation of a creator or constructor. We assume the execution of each statement is atomic. Objects are never destroyed:

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

3. Specifications of Types and Creators

3.1. Type Specifications

While we do not wish to endorse one formal specification method over another, we do assume a type specification includes the following information:

- The type's name;
- A description of the type's value space;
- For each of the type's methods:
  - Its name;
  - Its signature (including signaled exceptions);
  - Its behavior in terms of pre-conditions and post-conditions.

In our work we use formal specifications in the two-tiered style of Larch [14]. In the first tier, Larch *traits*, written in the style of algebraic specifications, are used to define a vocabulary of *sort* and *function* symbols. These symbols define a term language, where each term denotes a value (of a particular sort). For example, the term ''{ }'' might be used to denote the empty set value and the term ''$s \cup \{ i \}$'' might be used to denote the set value equal to the union of the set s and the singleton set { i }. Axioms and inductive rules of inference are used to determine when two terms, and hence two values, are equal. For example, such axioms would let us prove the commutativity of $\cup$ so that we could show that the two terms ''$s \cup \{ i \}$'' and ''$\{ i \} \cup s$'' denote the same set value. In our examples, we stick to standard notation for functions on sets, sequences, and tuples with their usual mathematical properties; in this paper we omit the Larch traits that would be used to specify such details. Many other specification languages like Z [26], OBJ3 [12], ACT-ONE [11] could just as easily be adopted to describe a type's value space.

In the second tier, Larch *interfaces* are used to describe the behaviors of an object's methods. For example, Figure 3-1 gives a specification for a set type whose objects have methods *insert, delete, select, elements*, and *equal*. The **uses** clause defines the value space for the type by identifying a sort. For example, the **uses** clauses in the figure indicates that all values of objects of type set are denotable by terms of sort S, as introduced in the MSet trait. The values of this particular sort are mathematical sets. The **methods** clause provides a specification for each method. Since a method's specifications need to refer to the method's object, we introduce a name for that object in the **for all** line. For each method, pre- and post-conditions are written in terms of **requires**, **modifies**, and **ensures** clauses. The pre-condition is the predicate of the **requires** clause; if this clause is missing, the pre-condition is trivially ''true.'' For example, *select's* pre-condition checks to see if the set has any elements. The post-condition is the

[1]This model is based on CLU semantics [19].

conjunction of the **modifies** and **ensures** clauses. *Insert's* post-condition says that the set's value may change because of the addition of its integer argument.[2] The **ensures** clause of the *elements* method makes use of the same_els function; we assume this is defined in the MSet trait (it returns true iff the set and the sequence contain exactly the same elements and furthermore the sequence has no duplicates).

set = **type**

  uses MSet (set **for** S)

  for all s: set

  methods

    insert = **proc** (i: int)
      modifies s
      ensures $s_{post} = s_{pre} \cup \{ i \}$

    delete = **proc** (i: int) **signals** (not_in)
      modifies s
      ensures if $i \in s_{pre}$ **then** $s_{post} = s_{pre} - \{i\}$
        else signal not_in

    select = **proc** ( ) **returns** (int)
      requires $s_{pre} \neq \{ \}$
      ensures result $\in s_{pre}$

    elements = **proc** ( ) **returns** (sequence[int])
      ensures same_els(s_{pre}, result)

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

Figure 3-1: A Type Specification for Sets

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 may change.

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.[3] 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) sets are identical. *Result* is a way to name a method's result parameter.

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. *Delete* signals

---

[2]Notice that we rely on the meaning of set union to guarantee that if i is already in the set, then inserting i again will not change the set's value.

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

the exception not_in if the element to be deleted is not in the set.

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)

Thus, a method's behavior is defined only when the pre-condition is satisfied; it is undefined otherwise.

The form of a Larch interface specification can easily be adapted for informal specifications. The **uses** clause is simply a description of the values, possibly using well-understood mathematical concepts. The specifications of the methods have **requires**, **modifies**, and **ensures** clauses, but the predicates are given informally (in English) in terms of the concepts introduced in the **uses** clause. We have used specifications like this with considerable success in our own work, and they are described in Liskov and Guttag [20].

3.2. Specifying Creators

Objects are created and initialized through creators. We do not include creators in type specifications so that different implementations of a type can have different creators and also so that subtypes can have different creators from their supertypes.

Figure 3-2 shows specifications for three different creators for sets. The first creator creates a new empty set, the second creates a singleton set, and the third creator allows the implementor to represent the set as a hash table, where the given integer argument determines the hash table's size. These examples show that different implementations may require different information when new objects are created, and therefore it is a good idea to allow the creators to be defined differently for them.

create = **proc** ( ) **returns** (set)
ensures $result_{post}$ = { } $\wedge$ **new**(result)

create_single = **proc** (i: int) **returns** (set)
ensures $result_{post}$ = { i } $\wedge$ **new**(result)

create_hash = **proc** (n: int) **returns** (set)
requires n > 0
ensures $result_{post}$ = { } $\wedge$ **new**(result)

Figure 3-2: Three Creators for Sets

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).

3.3. Type Specifications Need Explicit Invariants

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

To compensate for the lack of data type induction, we state the invariant explicitly in the type specification by means of an **invariant** clause. In the case that the invariant is trivial (i.e., identical to ‘‘true’’), as it is for sets, the **invariant** clause can be omitted. Figure 3-3 gives a specification of a bounded_set type for which the invariant is not trivial. The sort in this case is a pair <bound, els>, where bound is a natural number, and els is a mathematical set of integers. The invariant is that a bounded_set’s size never exceeds its bound. In general, the predicate $\phi(x_\rho)$ appearing in an **invariant** clause for type τ stands for the predicate $\forall$ x: τ, ρ: State . $\phi(x_\rho)$. We assume that components of the set’s value, e.g., bound, remain unchanged if not otherwise stated explicitly in the post-conditions of Figure 3-3.

bounded_set = **type**

  uses BSet (bounded_set **for** S)

  for all s: bounded_set

  invariant $| s_\rho.els | \leq s_\rho.bound$

  methods

    insert = **proc** (i: int)
      requires $| s_{pre}.els | < s_{pre}.bound$
      modifies s
      ensures $s_{post}.els = s_{pre}.els \cup \{ i \}$

    delete = **proc** (i: int) **signals** (not_in)
      modifies s
      ensures if $i \in s_{pre}.els$ **then** $s_{post}.els =$
        $s_{pre}.els - \{ i \}$ **else signal** not_in

    select = **proc** ( ) **returns** (int)
      requires $s_{pre}.els \neq \{ \}$
      ensures result $\in s_{pre}.els$

    elements = **proc** ( ) **returns** (sequence[int])
      ensures same_els(s_{pre}.els, result)

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

Figure 3-3: A Type Specification for Bounded Sets

Stating invariants explicitly in a type specification has three consequences. First, all creators for a type τ must establish τ’s invariant, I_τ:

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

Second, each method of the type must *preserve the invariant*. 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 *insert*, *delete*, *select*, *elements*, and *equal* each preserves the invariant for bounded_set. Informally the invariant holds because *insert's* pre-condition checks that there is enough room in the set for another element; *delete* either decreases the size of the set or leaves it the same; *select*, *elements*, and *equal* do not change the set at all. The proof ensures that methods deal only with legal values of an object's type.

Third, 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, we could prove that insertion on bounded_sets is commutative by appealing to the commutativity of $\cup$, which, as stated earlier, holds for all mathematical set values. Since the explicit invariant limits what invariant properties can be proved, the specifier needs to be careful when defining it.

However, it is important that the invariant be just strong enough and no stronger. A general rule is that specifications should be as weak as possible: they should state only what a user needs to depend on and nothing more. Weak specifications are desirable because they give more freedom to the implementor, and also, as we shall see, because they accommodate more subtypes. This rule affects the specifications of the methods as well as the invariant. For example, the specification of *select* is non-deterministic, leaving the implementor free to choose whatever element is most convenient in that particular implementation; it also permits us to define the fifo_set subtype discussed in the next section.

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.

4. Subtypes and the Subtype Relation

4.1. Specifying Subtypes

To state that a type is a subtype of some other type, we simply append a **subtype** clause to its specification. We allow multiple supertypes; there would be a separate **subtype** clause for each. Figure 4-1 gives an example. What distinguishes the fifo_set type from its set supertype is the more constrained behavior of its *select* method. In addition, it has an extra method, *remove*.

A subtype's value space may be different from its supertype's. Here, we use sequences (as defined in the Sequence trait) to represent values for fifo_set. The invariant indicates that only sequence values with no duplicate elements are legal values of fifo_set objects. (Again, we assume the no-duplicates function is defined appropriately in the sequence trait.) Under the **subtype** clause we define an *abstraction function*, A, that relates these sequence values to set values in the obvious manner. The **subtype** clause also lets specifiers rename methods of the supertype, e.g., *oldest* for *select;* all other methods of the supertype are ‘‘inherited’’ without renaming, e.g., *insert*, *delete*, *elements*, and *equal*. Any supertype method may be further constrained. In particular, *oldest* ensures that the returned element is the one that was first inserted longest ago. The additional *remove* method removes the oldest element from the set.

4.2. Subtype Relation Defined in Terms of Specifications

Various authors have defined subtyping by relating two type specifications [1, 15, 6, 3, 22]. The commonality among these subtype definitions is that they all capture the following two properties, stated informally:

- Values of the subtype relate to values of the supertype.
- Behaviors of the subtype methods relate to behaviors of corresponding supertype methods.

As introduced in the previous section, we use *abstraction functions* to relate value spaces. These are similar to America's transfer functions [1], Leavens's simulation relations [15], and Bruce and Wegner's coercion functions [3]. Since we include explicit invariants in type specifications we must make sure that the subtype definition requires that abstraction functions *respect the invariant*: the abstraction function must map legal values of the subtype to legal values of the supertype.

fifo_set = **type**

uses Sequence (fifo_set **for** Seq)

$\forall$ s: fifo_set

invariant no-duplicates(s_ρ)

methods

insert = **proc** (i: int)
  modifies s
  ensures **if** i in s_{pre} **then** $s_{pre} = s_{post}$
    else $s_{post} = s_{pre} \,||\, [\ i\ ]$

delete = **proc** (i: int) **signals** (not_in)
  modifies s
  ensures **if** ~(i in s_{pre}) **then signal** not_in
    else s_{post} = rem(s_{pre}, i)

oldest = **proc** ( ) **returns** (int)
  requires $s_{pre} \neq [\ ]$
  ensures result = first(s_{pre})

elements = **proc** ( ) **returns** (sequence[int])
  ensures same_els(s_{pre}, result)

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

remove = **proc** ( )
  modifies s
  ensures if $s_{pre} \neq [\ ]$ **then** s_{post} = rest(s_{pre})
    else $s_{pre} = s_{post}$

subtype of set (oldest **for** select)

A: Seq $\rightarrow$ S

$\forall$ q: Seq .
  A([ ]) = { }
  A(q || [i]) = A(q) $\cup$ { i }

Figure 4-1: A Type Specification for Fifo_set

For relating methods, we require that the predicate of the subtype method implies that of the supertype. (Recall that the predicate for method m is m.pre $\Rightarrow$ m.post.) This requirement guarantees that a method called on a subtype object will still exhibit acceptable behavior according to the corresponding supertype method's specification. This semantic requirement on the methods' pre- and post-conditions is analogous to the syntactic requirement on the methods' signatures [2, 24, 4] (see our contra/covariance rules below). Except for minor differences, our way of relating methods is similar to America's [1], Leavens's [15], and Cusack's [6].

More formally, given two types, σ and τ, each of whose specifications respectively preserves its invariant, I_σ and I_τ, we define the subtype relation, <, as follows:

<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. $I_\sigma(s) \Rightarrow I_\tau(A(s))$

   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.
2. Subtype methods preserve the supertype methods' behavior. Let m_τ of τ be the corresponding renamed method m_σ of σ (i.e., $R(m_\sigma) = m_\tau$). R may be partial, since it need not be defined on extra methods of σ, and must be onto and one-to-one. 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: σ:
     - *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.

Using this definition, it is a straightforward exercise to show that fifo_set is a subtype of set. Preserving the invariant is trivial because the invariant for sets is trivial. The reason that the *oldest* method simulates the behavior of *select* is that the specification of *select* does not constrain which element of the set is returned; the specification of *oldest* simply resolves this non-determinism in favor of the element that has been in the set the longest. The additional method, *remove*, causes no problem because the rule imposes no constraints on extra methods.

Our definition would not allow bounded_set to be a subtype of set. The pre-condition for set's *insert* is ‘‘true’’ since it is always legal to insert more elements in a set. Using our definition, we could not show that the pre-condition rule holds for *insert*. Intuitively, this restriction makes sense since a user of a set would be surprised if inserting did not cause the element to appear in the (unbounded) set. On the other hand, set could be a subtype of bounded_set since even though a user of a bounded_set might expect to reach the bound eventually, it is not surprising that any particular call has not reached the bound. However, the specification of bounded_set is a bit odd, because it is not possible for a user to observe the bound. If we were to change bounded_set's *insert* to signal an exception instead of assuming a non-trivial pre-condition, then by the exception rule we could show that set is not a subtype of bounded_set.

4.3. Adding Constraints to Type Specifications

The subtype rule given in the preceding section is perfectly adequate when procedures are considered in isolation and when there is no aliasing. For example, consider a procedure

get_max = **proc** (s: set) **returns** (int)
ensures $\forall$ x: Int . $x \in s_{pre} \Rightarrow result \geq x$

This procedure could be implemented by using the *elements* method to obtain the sequence of elements and then iterating over the sequence. In the execution of a call get_max(f), where f is a fifo_set, get_max actually calls subtype methods, but since each of these simulates the behavior of the corresponding supertype method, the procedure will execute correctly.

The definition is not sufficient, however, in the presence of aliasing, and also in a general computational environment that allows sharing of mutable objects by multiple users. Consider first the case of aliasing. The problem in this case is that within a single procedure a single object is accessible by more than one name, so that modifications using one of the names are visible when the object is accessed using the other name. For example, suppose S is a subtype of T and that variables

x: T
y: S

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

Now consider the case of an environment of shared mutable objects, such as is provided by object-oriented databases (e.g., Thor [21] and Gemstone [23]). (In fact, it was our interest in Thor that motivated us to study the meaning of the subtype relation in the first place.) In such systems, there is a universe containing shared, mutable objects and a way of naming those objects. In general, lifetimes of objects may be longer than the programs that create and access them (in fact, objects might be persistent) and users (or programs) may access objects concurrently and/or aperiodically for varying lengths of time. Of course there is a need for some form of concurrency control in such an environment. We assume such a mechanism is in place, and consider a computation to be made up out of atomic units (i.e., transactions) that exclude one another. The transactions of different computations can be interleaved and thus one computation is able to observe the modifications made by another.

Now let's consider the impact of having subtyping in such an environment. As an example, suppose one user installs an object that maps string names to sets. Later, a second user enters a fifo_set into that object mapped under some string name; a binding like this is analogous to assigning a subtype object to a variable of the supertype. After this, both users occasionally access that fifo_set object. The second user knows it is a fifo_set and accesses it using fifo_set methods. The question is: What does the first user need to know in order for his or her programs to make sense? We think it ought to be sufficient for a user to know only about the ‘‘apparent type’’ of the object. Thus the first user ought to be able to reason about his or her use of the fifo_set object using the invariant and history properties of set.

History properties are of course especially of interest for mutable types. In general 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. Notice that we implicitly quantify over all computations, c, and we do not require that ψ is the immediate successor of ρ. We can prove a history property by showing that it holds after the invocation of each method. Actually we only need to do the proof 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.

Our subtype rule lets us show that invariant properties of the supertype are preserved by its subtype. But what about history properties? Let us look at an example before we answer this question. Suppose we have a fat_set type that has all the methods of set except *delete*; fat_sets only grow while sets grow and shrink. The subtype relation discussed in Section 4.2 would allow us to prove that set is a subtype of fat_set. However, someone using a fat_set can deduce that once an element is inserted in the fat_set, it remains there forever. This history property does not hold for sets, and therefore, if the object in question is actually a set, and some other user is using it as a set, the user who views it as a fat_set will be able to observe surprising behavior (namely, the set shrinks). Therefore, set should *not* be a subtype of fat_set, and to disallow such subtype relations we need to extend our subtype definition.

To obtain a subtype relation that preserves history properties, we first add some information to specifications, a *constraint* clause that describes the history properties of the type explicitly[4]. In particular, we add to the specification for fat_set:

constraint $\forall\, x{:}\ \text{int} .\ x \in s_\rho \Rightarrow x \in s_\psi$

Similarly, we could add:

constraint $s_\rho.\text{bound} = s_\psi.\text{bound}$

to the specification of bounded_set to declare that a bounded_set's bound never changes. The predicate appearing in a **constraint** clause is an abbreviation for a history property. For example, fat_set's constraint expands to the following: For any computation, c,

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

Just as we had to prove that methods preserved the invariant, we must show that they *satisfy the constraint* by proving it for each mutator using the history rule. The constraint replaces the history rule as far as users are concerned: users can make deductions based on the constraint but they cannot reason using the history rule directly.

Next we extend our definition of the subtype relation to require that subtype constraints ensure supertype constraints. The full definition of our subtype relation is summarized in Figure 4-2, where C_σ and C_τ are the constraints given in the specifications of types σ and τ, respectively. We assume each type specification preserves invariants and satisfies constraints.

---

[4]The use of the term ‘‘constraint’’ is borrowed from the Ina Jo specification language [25], which also includes constraints in specifications.

Returning to our set and fat_set example, we see that the constraint rule is not satisfied since set has the trivial constraint ‘‘true,’’ which does not imply fat_set’s constraint. On the other hand, fifo_set is a subtype of set because it also has the trivial constraint.

As another example, consider a varying_set type that allows a bound to increase but does not provide any method to make such a change. It has the constraint:

constraint s_ρ.bound $\leq$ s_ψ.bound

Bounded_set is a subtype of this type, and so is a dynamic_set type with all the methods of varying_set plus *change_bound*, which increases the bound:

change_bound = **proc** (n: int)
  requires n $\geq$ s_{pre}.bound
  modifies s
  ensures s_{post}.bound = n

Intuitively, the non-determinism in the supertype is resolved in two different ways in the two subtypes: varying_set allows the bound to increase but does not require it; bounded_set does not take advantage of this opportunity, but dynamic_set does. This is another example of how subtypes can tighten the non-determinism present in the supertype.

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 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: σ:
     - *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: σ . $C_\sigma(x) \Rightarrow C_\tau[A(x_{pre})/x_\rho, A(x_{post})/x_\psi]$

Figure 4-2: Definition of the Subtype Relation

4.4. Discussion

Our first subtype definition (given in Section 4.2) is similar to others [1, 6, 15], but it does not go far enough. It fails to rule out subtype relations that would permit surprising behavior in the presence of shared mutable objects. Besides us, only Dhara and Leavens [9, 8] address this case. Rather than add explicit constraints in specifications, they place restrictions on the kinds of aliasing allowed in their programs. Their solution is limited to the special case of single-user, single-program environments.

We have also worked out another general approach that requires an *extension map* instead of explicit constraints [22]. This extension map is defined for all extra mutators introduced by the subtype and requires ‘‘explaining’’ the behavior of each extra mutator as a program expressed in terms of non-extra methods. For example, the *remove* method of fifo_set would be explained by the program:

i := s.oldest(); s.delete(i)

To show that history properties are preserved for non-extra mutators, we use the methods rule.[5] However, because the properties are not stated explicitly, they cannot be proved for the extra methods. Instead extra methods must satisfy any possible property, which is surely guaranteed if the extra methods can be explained in terms of the non-extra methods. Showing that the subtype constraint is stronger than the supertype's takes care of all the methods, not just the extra ones.

The approach using explicit constraints is appealing because it is so simple. In addition, it allows 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. Users are not permitted to use the history rule because if they did, they might be able to prove history properties that a subtype did not ensure. Therefore the specifier must be careful to define a strong enough constraint. In our experience the desired constraint is usually obvious, but here is an example of an inadequate constraint: Suppose the definer of fat_set mistakenly gave the following constraint:

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

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

5. Conclusions

This paper makes two contributions. First it provides a specification technique for object types that allows creators to be specified separately from types. Separating the creators is important because it allows different implementations of a type to have different creators, and also because it allows subtypes to have different creators from supertypes. However, leaving out creators leads to the loss of data type induction, which is needed to prove type invariants. We make up for this loss by including explicit invariants in our specification. Our specifications also contain explicit constraints; these identify a minimal set of history properties that methods of the type and all its subtypes must preserve.

Our specifications are based on Larch; we believe that this is a particularly readable and easy-to-use approach for programmers. In addition, it is easy to give informal specifications that have the same general form as our formal ones. In our experience, informal specifications that have a prescribed format

[5] A more constraining methods rule that requires subtype methods to have identical pre-conditions to those of the corresponding supertype methods is needed. Thanks to Ian Maung (private communication) for pointing this out.

and information content are very useful in program development.

Our second contribution is a new definition of the subtype relation. We argue that all properties provable about objects of a supertype should also hold for objects of its subtypes. This very strong definition ensures that if one user reasons about a shared object using properties that hold for its apparent type, that reasoning will be valid even if the object actually belongs to a subtype and is manipulated by other users using the subtype methods. Our definition of subtyping highlights the importance of history properties by making constraints explicit in specifications and by requiring that subtypes guarantee supertype constraints. Explicit constraints allow us to give a simple and straightforward definition of subtyping that works even in a very general environment in which possibly concurrent users share mutable objects.

This paper showed how these two contributions interact with each other. In the presence of subtyping, type specifications have to change to accommodate the separation of creators from an object's other methods. At the same time, type specifications must contain sufficient information for users to prove that one type is a subtype of another.

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 helpful verbal and e-mail discussions on subtyping and pointers to related work. In addition, Gary, John Guttag, Mark Day, Sanjay Ghemawat, and Deborah Hwang, Greg Morrisett, Eliot Moss, Bill Weihl, Amy Moormann Zaremski, and the referees gave useful comments on earlier versions of this paper.

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

References

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

2. Black, A. P., Hutchinson, N., Jul, E., Levy, H. M., and Carter, L. "Distribution and Abstract Types in Emerald". *IEEE TSE SE-13* (Jan. 1987), 65-76.

3. Bruce, K. B., and Wegner, P. An Algebraic Model of Subtypes in Object-Oriented Languages (Draft). ACM SIGPLAN Notices, Oct., 1986. Object-Oriented Programming Workshop.

4. Cardelli, L. "A semantics of multiple inheritance". *Information and Computation 76* (1988), 138-164.

5. Carrington,, D., Duke, D., Duke, R., King, P., Rose, G., and Smith, P. Object-Z: An Object Oriented Extension to Z. FORTE89, International Conference on Formal Description Techniques, Dec., 1989.

6. Cusack, E. Inheritance in object oriented Z. Proceedings of ECOOP '91, 1991.

7. Cusack, E., and Lai, M. Object-Oriented Specification in LOTOS and Z, or My Cat Really is Object-Oriented! Foundations of Object Oriented Languages, June, 1991, pp. 179-202. LNCS 489.

8. Krishna Kishore Dhara. Subtyping among mutable types in object-oriented programming languages. Master Th., Iowa State University, Ames, Iowa,1992.

9. Krishna Kishore Dhara and Leavens, G. T. Subtyping for mutable types in object-oriented programming languages. Tech. Rept. TR #92-36, Department of Computer Science, Iowa State University, Ames, Iowa, Nov., 1992.

10. Duke, D., and Duke, R. A History Model for Classes in Object-Z. Proceedings of VDM '90: VDM and Z, 1990.

11. Ehrig, H., and Mahr, B. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, 1985.

12. Goguen, J. A., Kirchner, C., Kirchner, H., Megrelis, A., Meseguer, J., and Winkler, T. An Introduction to OBJ3. Proceedings, Conference on Conditional Term Rewriting, 1988, pp. 258-263. LNCS 308.

13. Goguen, J. A., and Meseguer, J. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. Research Directions in Object Oriented Programming, 1987.

14. Guttag, J. V., Horning, J. J., and Wing, J. M. "The Larch Family of Specification Languages". *IEEE Software 2*, 5 (sept 1985), 24-36.

15. Leavens, G. Verifying Object-Oriented Programs That Use Subtypes. Tech. Rept. 439, MIT Lab. for Computer Science, Feb., 1989. Ph.D. thesis.

16. Leavens, G. T. "Modular specification and verification of object-oriented programs". *IEEE Software 8*, 4 (July 1991), 72-80.

17. Leavens, G. T., and Krishna Kishore Dhara. A Foundation for the Model Theory of Abstract Data Types with Mutation and Aliasing (preliminary version). Tech. Rept. TR #92-35, Department of Computer Science, Iowa State University, Ames, Iowa, Nov., 1992.

18. Leavens, G. T., and Weihl, W. E. Reasoning about Object-Oriented Programs that use Subtypes. ECOOP/OOPSLA '90 Proceedings, 1990.

19. Liskov, B. et al. *CLU Reference Manual.* Springer-Verlag, 1981.

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

21. Liskov, B. Preliminary Design of the Thor Object-Oriented Database System. Proc. of the Software Technology Conference, April, 1992. Also Programming Methodology Group Memo 74, MIT Laboratory for Computer Science, Cambridge, MA, March 1992.

22. Liskov, B., and Wing, J. M. A New Definition of the Subtype Relation. Proceedings of ECOOP '93, Kaiserslautern, Germany, 1993. to appear.

23. Maier, D., and Stein, J. Development and Implementation of an Object-Oriented DBMS. Readings in Object-Oriented Database Systems, 1990, pp. 167-185.

24. Schaffert, C., Cooper, T., and Wilpolt, C. Trellis: Object-Based Environment Language Reference Manual. Tech. Rept. 372, Digital Equipment Corp./Easter Research Lab., 1985.

25. Scheid, J., and Holtsberg, S. Ina Jo Specification Language Reference Manual. Tech. Rept. TM-6021/001/06, Paramax Systems Corporation, A Unisys Company, June, 1992.

26. Spivey, J. M. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

27. Utting, M. *An Object-Oriented Refinement Calculus with Modular Reasoning*. Ph.D. Th., University of New South Wales, Australia, 1992.