Subtyping for Distributed Object Stores
(Extended Abstract)

Jeannette M. Wing

April 1997
CMU-CS-97-121

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This extended abstract will appear in the Proceedings of the Second IFIP International Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS), July 1997.

Abstract

I review the Liskov and Wing subtype definition that takes into consideration the problem of subtyping in the presence of mutable objects. I then show how this notion of subtyping is relevant to the design of the TOM object repository whose main application today is a data type conversion service accesssible through the Web.

This research is sponsored in part by the Defense Advanced Research Projects Agency and the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, F33615-93-1-1330, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031 and in part by the National Science Foundation under Grant No. CCR-9523972. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency Rome Laboratory or the U.S. Government.

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

1 Introduction

The programming language community has come up with many definitions of the subtype relation. The goal is to determine when this assignment

x: $T := E$

is legal in the presence of subtyping. 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.

The question of when is one type a subtype of another is especially tricky to answer in the presence of shared mutable objects. For example, in the context of programming languages, we have the common situation where during the course of executing a program, two or more pointers reference the same object in the heap. A change to the object accessed by one pointer will be reflected in any further access to that object made through the other pointers. In Fig. 1, for example, x and y are pointers of type T and subtype S, respectively, that refer to the same object; this aliasing means that a change made through y will be visible through x.

Relevant to this workshop's theme, the above aliasing situation is a special case of what happens in a distributed environment where objects are stored in persistent repositories. Generalize the notion of "pointer" to a notion of "handle," e.g., an index entry of a persistent database, a file name in a distributed file system, or a URL for the Web. Generalize the notion of a programming language's run-time heap to the notion of a persistent object store or a distributed file system. Generalize the notion of multiple uses of an object during the execution of a single program to the notion of multiple users (or equivalently to multiple independent programs) that share access to the object. The situation in a distributed environment is more general since unlike a program's heap, objects live indefinitely, and do not disappear when the program terminates. Moreover, unexpected behavior that can result from one user changing an object with respect to another user's viewpoint is more likely since users may be unaware of each other's existence.

Programmers make two kinds of changes to a supertype definition when defining a subtype: they add new methods and they change old methods of the supertype. Unconstrained, however, both kinds of modifications can can lead to surprising behavior. Consider a type fat_set that has only *create*, *insert*, and *size* methods. If we were to define a subtype, set, by adding a new method, *delete*, then suddenly the fact that a fat_set object can only grow in size no longer is true, surprising users who think they have a fat_set object when it really is a set object. So, we cannot just add methods willy-nilly. Similarly, consider a plain_elephant type that has just one method, *get_color*, which always returns gray. If we were to define a subtype royal_elephant and correpondingly change the behavior (e.g., through overriding) of *get_color* to return blue, then users who think they have a plain_elephant object may see later that its color has changed. So, we cannot just change methods willy-nilly either.

What we need is a subtype requirement that constrains the behavior of subtypes so that users will not encounter any surprises:

> *No Surprises Requirement*: Properties that users rely on to hold of an object of a type T should hold even if the object is actually a member of a subtype S of T.

The property users rely on for fat_sets is that they only grow and never shrink in size; the property users rely on for plain_elephants is that their color is always gray.

In their 1994 TOPLAS paper "A Behavioral Notion of Subtyping" Liskov and Wing [8] addressed the problem of what subtyping means, especially in the presence of shared mutable objects. They provide two alternative definitions in their paper. In this extended abstract, I summarize only one of these definitions, to highlight their main points. I then describe an object repository that the TinkerTeach Project built at Carnegie Mellon and use it to explain what relevance this subtyping notion has in practice.

2 Review of the Liskov and Wing Notion of Subtyping

Key to understanding the Liskov and Wing notion of subtyping is the use of the specification of an object's type. Determining when one type is a subtype of another is based on showing that certain properties hold

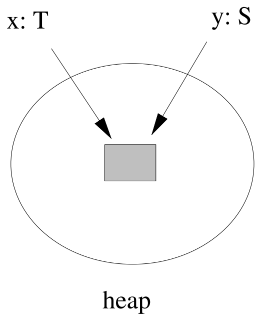

Figure 1: Shared Access to Mutable Objects in a Heap

between the two type specifications.

2.1 Type Specifications

A type specification contains the following information:

- The type's name.
- A description of the set of values over which objects of the type ranges.
- For each of the type's methods:
  - Its name.
  - Its signature, i.e., the types of its arguments (in order), result, and signaled exceptions.
  - Its behavior in terms of pre-conditions and post-conditions.
- A type constraint.

Fig. 2 gives an example of a type specification for bounded bags. To the spirit of the theme of this workshop, I give formal specifications, written in the style of Larch [6], but I could just as easily have written informal specifications. Since these specifications are formal we can do formal proofs, possibly with machine assistance like with the Larch Prover [5], to show that a subtype relation holds [10].

The BBag Larch Shared Language trait and the **invariant** clause together describe the set of values over which bag objects can range. The **requires**, **modifies**, and **ensures** clauses specify the methods' pre- and post-conditions. The **constraint** clause specifies the type constraint.

A *type invariant* constrains the value space for a type's objects. In the bag example, the type invariant says that the number of integers stored in a bag in any state, ρ, is always less than or equal to the bag's (fixed) bound.

To ensure that the specification is *consistent*, the specifier must show that (1) each constructor of an object of the type establishes the invariant and (2) each of the type's methods preserves it. (For methodologically reasons, Liskov and Wing specify constructors separately from the other type's methods so none are shown in Fig. 2.)

The inclusion of pre- and post-conditions in the specification of a type's methods allows us to relate the two types' behaviors; this is the main difference between the Liskov and Wing definition of subtyping and those that rely on just signature information (e.g., Cardelli [2]). For example, two methods with the same signature (e.g., *get* and *card* for bags) may have dramatically different behavior. Relying on just signature bag = **type**

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

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

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

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

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

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

end bag

Figure 2: A Type Specification for Bags

information identifies these methods that behave differently; thus, finer subtyping distinctions can be made when behavioral information is used in addition to signature information.

The inclusion of the *type constraint* is what distinguishes the Liskov and Wing work from all others (e.g., America [1], Cusack [3], Leavens [4, 7]) that also include some kind of behavioral information. To foreshadow what is coming in the next section: Not only must a supertype's type invariant and methods be preserved by the subtype's, but so must its type constraint.

The type constraint is intended to capture certain kinds of properties of an object that are required to remain invariant over the indefinite lifetime of the object. Liskov and Wing call them *history properties*. For example, if a window's size, an elephant's color, or a bag's bound is to remain constant, then these properties should be stated as type constraints; if the size of a set can only grow and never shrink or a counter's value only monotonically increases, then these properties should be stated as type constraints.

Stating history properties through type constraints is exactly how Liskov and Wing deal with mutable objects. Formally, a type constraint is a two-state predicate, $C(\rho, \psi)$, where ρ and ψ are any two successive states in any computation. A type constraint is similar to a type invariant except a type invariant is a single-state predicate–a property that holds in every state, ρ, of a computation. Since a type constraint is a property relating two successive states, it captures what behavior may *not* change from state to state; hence it captures additional "invariant" properties of mutable objects.

2.2 The Subtype Relation

The subtype relation is defined in terms of a checklist of properties that must hold between the specifications of the two types, S and T. Since in general the value space for objects of type S will be different from the value space for those of type T we need to relate the different value spaces; we use an *abstraction function*, A, to define this relationship. Also since in general the names of the methods of type S can be different from those of type T we need to relate which method of S corresponds to which method of T; we use a *renaming* map, R, to define this correspondence. (In a programming language like Java, this is just the identity map, as realized though method overloading.)

S is a *subtype* of T if the following three conditions hold (informally stated):

1. The abstraction function respects the invariants. If the subtype invariant holds for any subtype value, s, then the supertype invariant must hold for the abstracted supertype value $A(s)$.

2. Subtype methods preserve the supertype methods' behavior. If m is a subtype method then let n be the corresponding $R(m)$ method of the supertype.

   - Arguments to m are contravariant to the corresponding arguments to n; m's result is covariant to the result of n.
   - Any exception signaled by m is contained in the set of exceptions signaled by n.
   - n's pre-condition implies m's and m's post-condition implies n's (under the abstraction function).

3. Subtype constraints ensure supertype constraints. S's type constraint implies T's type constraint (under the abstraction function).

Why does this subtype relation guarantee that the No Surprises Requirement holds? Recall that the Requirement refers vaguely to "properties." What this definition of subtype guarantees is that certain properties of the supertype–those that are stated explicitly or provable from a type's specification–are preserved by the subtype. The first and third conditions directly relate the invariant and history properties; the second condition relates the behaviors of the individual methods, and thus preserves any observable behavioral property of any program that invokes those methods. Though the Liskov and Wing paper focuses on only these kinds of behavioral properties, it would be reasonable to extend this definition to other kinds of properties by extending the scope of what is included in a type specification.

Consider the type family of bags in Fig. 3. The constraint for a supertype, varying_bag, of the bag type given in the previous section is that the bound may change or stay the same. Another subtype of varying_bag could be dynamic_bag, with the trivial contraint true (thus saying its bound may change). Dynamic_bag would have an additional method, *change_bound*,

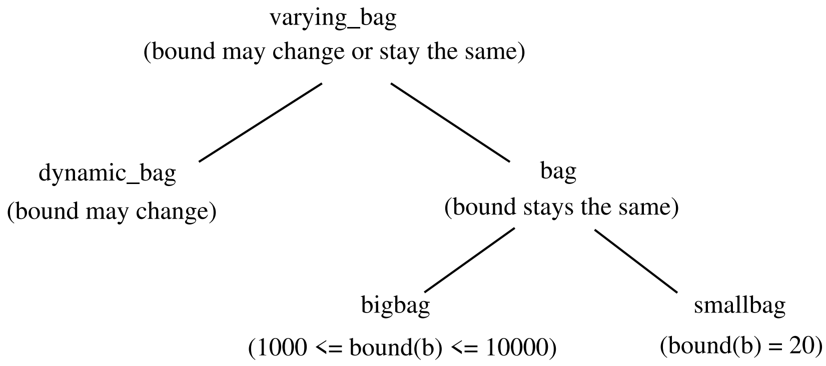

Figure 3: A Type Family for Bags

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

which would not make sense for the bag type to have.

Further subtyping the bag type we can, for example, define bigbag with a constraint that its bound be within a certain range and smallbag with a constraint that its bound be fixed to be 20.

This definition of subtyping supports multiple supertypes. If S is a subtype of both T and U, then the designer is obligated to show the above checklist of conditions holds between S and T and between S and U. Implementation problems that arise because of multiple inheritance are irrelevant; subtyping is a relationship between specifications, not implementations.

3 An Example Object Repository: The TOM Server

How does understanding the subtype relation help the system designer? In the second half of this extended abstract I describe an object repository that the Carnegie Mellon TinkerTeach Project built and its key design principle. This project was done independently of the work on subtyping, but in retrospect two lessons can be learned from the implementor's design decision as related to the Liskov and Wing notion of subtyping:

- Avoiding mutability simplifies defining a subtype hierarchy.
- Mutability cannot be completely avoided in an open distributed environment.

Before I explain these seemingly inconsistent statements, I describe the object repository's functionality.

As part of his Ph.D. thesis, John Ockerbloom invented a Typed Object Model [9], a data model involving objects, types, and their associated metadata. He implemented an instance of this model, a TOM server, which currently supports the ability for users in a distributed environment to store data types and data conversion functions, to register new ones, and to find existing ones. The kinds of data types TOM supports today are different kinds of document types (e.g., Word, LaTex, PowerPoint, binhex, html) and "packages" of such document types (e.g., a mail message that has an embedded postscript file, a tar file, or a zip file). The kinds of data conversions TOM supports are off-the-shelf converters like *postscript2pdf* (i.e., AdobeDistiller^{TM}), off-the-Web ones like *latex2html*, and some home-grown ones like *powerpoint2html*.[1] As

[1] The conversion of a PowerPoint document to an .html file is actually done through the application of nine different intermediate steps going through different intermediate types like rtf, postscript, and ppm. These intermediate converters do things like converting postscript files to ppm files, resizing and rotating ppm files, and converting ppm files to gif files. Users see none of these intermediate steps.

of mid-March, 200 sites from over 20 countries in 6 continents have accessed TOM.[2]

Today a specific application of TOM is to handle type conversion tasks, which includes a Web-based user interface built for the TinkerTeach Project. This user interface hides much of the complexity of type conversion from the user, in three ways:

- TOM is a system of *type brokers*. If a user makes a request to one instance of a TOM server, S, and S does not know about the data type or converter in question, but does know of another instance, T, that does, then completely transparently to the user, S will contact T to process the request. Thus, there can be multiple instances of a TOM server where each knows about a few types and converters; collectively all the TOM servers comprise a distributed object server.
- TOM can compose converters to do conversions. Given a source type and a target type (by the user), TOM can figure a plan of conversion steps to apply. It can make such plans on the fly, such as when it composes an *rtf2html* converter with an *html2text* converter.
- Given an object (e.g., a Word document) to convert, TOM uses heuristics to guess what the type of the source object is. It can also tell a user when a requested conversion is unsupported or meaningless.

3.1 Simplify Life: Avoid Mutability

When Ockerbloom originally designed TOM, he made the following critical design decision:

All objects are immutable.

The rationale behind the decision is that he wanted to treat arbitrary information in a distributed environment like the Web as objects. If objects can change in value, then issues of storage, update, and concurrency control must be resolved, perhaps using standard distributed file system or distributed database techniques. If objects cannot change in value, then TOM does not have to worry about how they are stored, where they are stored, how they are updated, if and how they are copied or replicated, and how to coordinate concurrent access to them. Rather, objects can live anywhere, be created by anyone, and be shipped around freely.

In principle, by deciding that no object can be mutable, Ockerbloom is able to avoid the problem of shared access to mutable objects. Thus, TOM does not have to worry about what subtyping means in the presence of mutability. As a corollary, showing the subtype relation holds of TOM's type hierarchy is simpler since showing constraints are preserved is trivial.

TOM does support an interesting subtype hierarchy. Figure 4 gives a subgraph of the TOM type hierarchy. For example, TOM makes a distinction between a package type that has clear delimiters (delimited_package) and one which is just a mail_message, which contains a mail header and some uninterpreted contents. A parsed_mail_message is distinct from a mail_message because the type of the message's contents has been determined (e.g., a postscript file). Also TOM supports packages of packages, and so for example, a mail message can contain a forwarded mail message which itself contains a MIMEmultipart file; TOM is "smart" enough to unwrap these packages and present their contents in a way that users can meaningfully interact with the individual pieces.

Notice two examples of multiple supertypes in this subgraph. The parsed_mail_message type is a subtype of both mail_message and delimited_package, and a mail_message itself is a subtype of both package and communication.

Ockerbloom carefully designed his type hierarchy so that each subtype either only adds new methods or changes (by overriding) old methods in a constrained way. Thus, proving that the Liskov and Wing subtype conditions hold for the TOM hierarchy is relatively easy:

- If no changes to old methods are made then the proof is trivial. Since each object is immutable, neither old nor new methods can be mutators, so constraints are trivially preserved. Since no old method is overridden, invariants are preserved and the behavior of old methods is preserved. In the typical case, the subtype object simply has more state information, e.g, extra attributes, and the abstraction

[2] I use it on a daily basis. The Web site is: `http://tom.cs.cmu.edu/`.

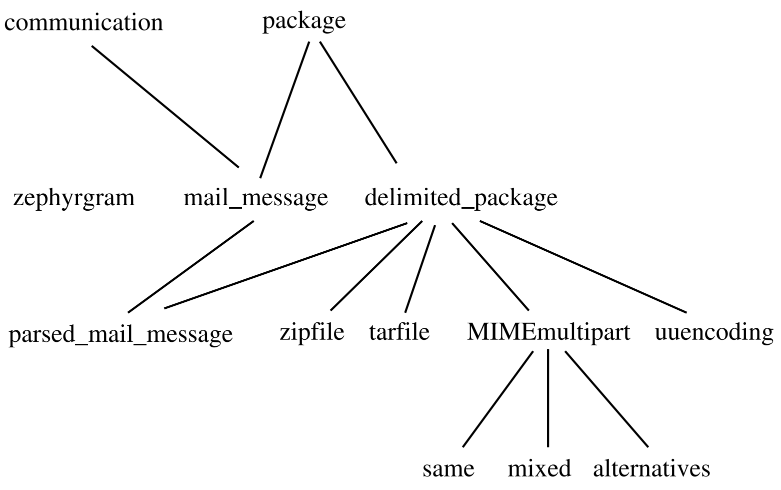

Figure 4: Part of TOM's Type Hierarchy

function is the obvious many-to-one function that throws away the extra state information. In their paper, Liskov and Wing call these *extension* subtypes since the subtype extends the supertype by providing additional methods and possibly additional state.

- If changes to old methods are made, then Part 2 of the subtype definition applies: the contra/covariant rules, the exception rule, and most importantly the pre-/post-condition rules must be shown. If subtypes always only further constrain the behavior of the corresponding supertype methods, then it is easy to show that invariants and constraints are preserved (given that Part 2 holds and that the specifications are consistent). In their paper, Liskov and Wing call these *constrained subtypes* because the degree of nondeterminism is reduced in the subtype. The abstraction function is usually into rather than onto.

3.2 Life is Not So Simple: Context Matters

Since TOM's objects are all immutable, TOM can treat the values of these objects as the real "objects." In other words, from TOM's view there is a huge value space, where each value can be considered an "object" (an entity that provides a set of methods in the traditional sense). TOM also supports *handles* to objects, e.g., file names and URLs; handles are also TOM immutable objects and provide a *dereference* method. For example, the contents of a file is a TOM object, not the file itself; a file name is a TOM object, which when dereferenced, refers to the contents of a file.

By definition values of immutable objects cannot change. However, the *binding* between handles to values/TOM objects may change and *TOM has no control over this binding.* In particular, TOM's environment can change the binding between handles and values, and so from the user's viewpoint, it looks as if these objects are mutable. In other words, the dereference method on a handle might yield different results at different times, such as when someone has edited the file being referenced. And, two different handles, e.g., two different URLs, may dereference to the same file contents. TOM cannot control or even know about this binding. For example, it is common for many different URLs to refer to the same file on a given Web site and it is common for system administrators to export a URL for remote access but use an internal file name for local access. Thus, from a more global perspective, TOM objects are shared and these objects are mutable. We are back to our original problem: what does subtyping mean in the presence of shared mutable objects?

Unfortunately, as users of local and distributed file systems, the Web, or publicly accessible persistent object repositories, we have no control over the semantic guarantees that these different contexts provide. Unix-like file systems, for example, provide no consistency guarantees; a change by one user to a file may not be seen by another who has a replica or cached copy of that file. These weak consistency guarantees mean that while the subtyping relation may hold from TOM's internal viewpoint, it can be intentionally or inadvertently violated by someone who accesses a TOM object from outside of TOM, by implicitly changing the binding between some handle and TOM object.

This situation is neither new nor surprising. For any persistent object repository that does not sit in isolation, i.e., makes its objects available through means other than that repository's interface, the same situation will arise. Thus, this situation simply serves as a warning to the user of that persistent object repository and as a reminder to its designer: Context matters and must be taken into consideration when accessing the repository's objects.

4 Summary

I reviewed the Liskov and Wing subtype definition that takes into consideration the problem of subtyping in the presence of mutable objects. The key ideas behind their notion of subtyping are (1) to consider the behavior of objects, as specified through pre-/post-conditions for methods, and invariants and constraints for data types; and (2) to consider *history properties*, as captured by type constraints.

I also showed how this notion of subtyping is relevant to the design of the TOM object repository whose main application today is a data type conversion service. While TOM views all its objects as immutable, its environment may indeed provide alternative means of access to these objects and thus users may make changes to them. In TOM, this situation is realized by changing the binding between a handle and TOM object; TOM has no control this binding. If users make such changes then they need either to ensure that the changes are consistent with the behavior specified for the objects' types or to realize that the No Surprises Requirement can be violated.

5 Acknowledgments

I thank Barbara Liskov, my co-author on previous papers on this subject. Parts of this abstract, in particular the opening paragraph, the statement of the No Surprises Requirement, the bag specification, and the bag type family, are taken from our TOPLAS paper. I would also like to thank John Ockerbloom for building TOM and his constructive comments on a draft of this abstract. Finally, I thank all the TinkerTeach Project members, in particular, Norm Papernick, our staff programmer, who maintains the TOM conversion service.

References

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

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

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

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

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

[6] J.J. Horning, J.V. with S.J. Garland Guttag, K.D. Jones, A. Modet, and J.M. Wing. *Larch : Languages and Tools for Formal Specification.* Springer-Verlag, New York, 1993.

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

[8] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, November 1994.

[9] John Ockerbloom. Exploiting structured data in wide-area information systems. Technical Report CMU-CS-95-184, Carnegie Mellon Computer Science Department, Pittsburgh, PA, 1995.

[10] Amy M. Zaremski. Signature and specification matching. Technical Report CS-CMU-96-103, CMU Computer Science Department, January 1996. Ph.D. thesis.