Respectful Type Converters

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

May 7, 1998

Abstract

In converting an object of one type to another, we expect some of the original object's behavior to remain the same, and some to change. How can we state the relationship between the original object and converted object to characterize what information is preserved and what is lost after the conversion takes place? We answer this question by introducing the new relation, *respects*, and say that a type converter function $C : A \rightarrow B$ *respects* a type T. We formally define *respects* in terms of the Liskov and Wing behavioral notion of subtyping; types A and B are subtypes of T.

We explain in detail the applicability of respectful type converters in the context of the Typed Object Model (TOM) Conversion Service, built at Carnegie Mellon and used on a daily basis throughout the world. We also briefly discuss how our *respects* relation addresses a similar question in two other contexts: type evolution and interoperability.

1 Motivation

The tremendous growth of the Internet and the World Wide Web gives millions of people access to vast quantities of data. While users may be able to retrieve data easily, they may not be able to interpret or display retrieved data intelligibly. For example, when retrieving a Microsoft Word document, without a Microsoft Word program, the user will be unable to read, edit, display, or print it. In general, the type of the retrieved data may be unknown to the retrieving site.

Users and programs cope with this problem by *converting* data from one type to another, e.g., from the unknown type to one known by the local user or program. Thus, to view the Word document, we could convert it to ASCII text or HTML, and then view it through our favorite text editor or browser. A picture in an unfamiliar Windows bitmap type could be converted into a more familiar GIF image type. A mail message with incomprehensible MIME attachments could be converted from an unreadable MIME-encoded type to a text, image, or audio type that the recipient could examine directly. In general, we apply *type converters* on (data) objects, transforming an object of one type to an object of a different type.

1.1 What Information Do Type Converters Preserve?

In converting objects of one type to another we expect there to be some relationship between the original object and the converted one. In what way are they similar? The reason to apply a converter in the first place is that we expect some things about the original object to change in a way that we are willing to forgo,

*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.

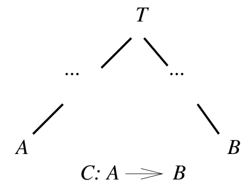

Figure 1: Does Converter C Respect Type T?

but we also expect some things to stay the same. For example, suppose we convert a LaTeX file to an HTML file. We may care to ensure that the raw textual contents of the original LaTeX document are preserved, but not the formatting commands since they do not contribute to the meaning of the document itself; here the preserved information is the underlying semantics of the text contained in the document. Alternatively, if we convert a LaTeX file to a table-of-contents document, we may care to ensure that the number, order, and titles of chapters and sections in the original document are preserved, but not the bulk of the text; here the preserved information is primarily the document's structure.

The question we address in this paper is "How can we characterize what information is preserved by a type converter?" Our answer is given in terms of the behavior of some type T. Informally, we say a *converter* $C : A \to B$ *respects type* T if the original object of type A and the converted object of type B have the same behavior when both objects are viewed as a type T object. That is, from T's viewpoint, the A and B objects look the same. If the converter respects a type, then it preserves that type's observable behavior. This paper formalizes this novel notion of *respectful type converters.*

Our particular formalization of *respects* exploits the *subtype* relationship that holds among types of objects. The Liskov and Wing notion of behavioral subtyping [9] conveniently characterizes semantic differences between types. If S is a subtype of T, users of T objects cannot perceive when objects of type S are substituted for T objects. Intuitively, if C respects type T, an ancestor of both A and B in the subtype hierarchy, then T captures the behavioral information preserved by C.

This paper spells out how to determine whether a given ancestor T in a type hierarchy is respected by a converter $C : A \to B$ (Figure 1). In general, A and B need not be subtypes of each other; in practice, they are often siblings or cousins in a given type hierarchy. Also, in general, T is not necessarily the least common ancestor of A and B.

Here is an example of why T is not just any ancestor of A and B. Suppose that the PNG image and GIF image types are both subtypes of a pixel_map type (Figure 2) that specifies the colors of the pixels in a rectangular region. GIF images are limited to 256 distinct colors; PNG images are not. Assuming the pixel_map type also does not have a color limit, then a general converter from PNG images to GIF images would not respect the pixel_map type: it is possible to use pixel_map's interface to distinguish a PNG image with thousands of colors from its conversion to a GIF image with at most 256 colors. On the other hand, suppose pixel_map is in turn a subtype of a more generic bitmap type that simply records whether a graphical element is set or clear (see Figure 2). Suppose further that elements in a pixel_map are considered set if they are not black, and clear if they are black. As long as the PNG to GIF converter does not change any non-black color to black (or black to non-black), and otherwise preserves the pixel layout, there is no way for the bitmap interface to distinguish the PNG image from the GIF image that results from the conversion. Here then, the PNG to GIF converter respects the bitmap type.

Although we explore the idea of respectful type converters primarily in the context of file and document converters, type converters show up in other contexts. Most programming languages have built-in type converters defined on primitive types, for example, *ascii2integer*, *char2string*, and *string2array[char]*. The real world is continually faced with painful, costly, yet seemingly simple conversions: the U.S. Postal System converted five-digit zip codes to five+four-digit zip codes; Bell Atlantic recently added a new area code necessitating the conversion of a large portion of phone numbers in Western Pennsylvania from the 412 area code to 724; and payroll processing centers routinely need to convert large databases of employee records

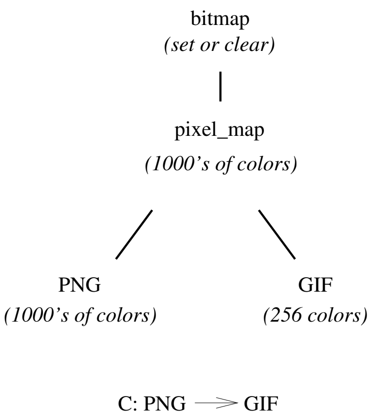

Figure 2: The PNG to GIF converter respects bitmap, but not pixel_map.

whenever extra fields are added to the relevant database schema.

1.2 Roadmap to Rest of Paper

In this paper we formally characterize the notion of when a converter *respects* a type. We first review in Section 2 how we specify types and determine when one type is a subtype of another, borrowing directly from Liskov and Wing's behavioral notion of subtyping [9]. In Section 3 we exploit this notion of subtyping to define the *respects* relation between a converter and a type.

Armed with these formal definitions, in Section 4 we show how they are used in practice in the application of the Typed Object Model (TOM) Conversion Service, which we built at Carnegie Mellon and use on a daily basis. In Section 5 we discuss a further extension of *respects* that enables us to relate the implementation of types to their specifications; specifically, the abstraction function used to show the correctness of the implementation of abstract types fits neatly into the way we define *respects*. In Section 6 we briefly discuss two other applications of our *respects* relation: type evolution and reusability in the context of interoperability. Section 7 gives a status report of the TOM Conversion Service and puts it in a broader setting given TOM's environment. We close with a discussion of future work.

2 Behavioral Subtyping

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.

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.

which guarantees Liskov's *substitutability* principle of subtypes [8].

In their 1994 TOPLAS paper "A Behavioral Notion of Subtyping" Liskov and Wing [9] formalized this requirement in their definition of subtyping. The novel aspect of their subtype definition is the ability to handle mutable types. In this paper, we present a simplified version of their definition since for all our applications we can ignore mutable objects. We discuss the extension of our work to mutability as future work in Section 8.

Under this simplification, first we describe our model of objects and types and then how we specify types. Finally we define the subtype relation.

2.1 Model of Objects, Types, and 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 or observe that object.

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

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

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

We model a type as a triple, $\langle O, V, M\rangle$, where $O \subseteq \mathit{Obj}$ 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* or an *observer*. Constructors of an object of type τ return new objects of type τ; observers return results of other types. Objects are *immutable*: their values cannot change. All our types are immutable. 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*.

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

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

Each transition, Tr_i, of a computation sequence is a partial function on states.

Objects are never destroyed: $\forall\, 1 \leq i \leq n \,.\, dom(\rho_{i-1}) \;\subseteq\; dom(\rho_i)$.

2.2 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.

GIF: **type**

uses GIFImage (gif **for** G)
for all g: GIF
invariant $\mid colorrange(g_\rho) \mid \leq 256$

color *get_color* (*i, j:* int)
ensures $result = overlay(g, i, j)$

frame *get_frame* (*i:* int)
requires $1 \leq i \leq len(g)$
ensures $result = g[i]$

bool *is_set* (*i, j:* int)
ensures $result = (overlay(g, i, j) \neq \text{BLACK})$

bool *equal* (*a:* GIF)
ensures $result = (a = g)$

end GIF

Figure 3: A Larch Type Specification for GIF Images

Figure 3 gives an example of a type specification for GIF images. We give formal specifications, written in the style of Larch [7], but we 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 [4], to show that a subtype relation holds [11].

The GIFImage Larch Shared Language trait and the **invariant** clause in the Larch interface type specification for GIF images together describe the set of values over which GIF image objects can range. GIF images are sequences of frames where each frame is a bounded two-dimensional array of colors. Appendix A contains the GIFImage trait as well as those for frame sequences, frames, colors, etc.

A type invariant constrains the value space for a type's objects. In the GIF example, the type invariant says that a GIF image can have at most 256 different colors. (The *colorrange* function defined in GIFImage returns the range of colors mapped onto by the array.) The predicate $\phi(x_\rho)$ appearing in an **invariant** clause for type τ stands for the predicate: For all computations, c, and all states ρ in c:

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

The **requires** and **ensures** clauses in the Larch interface specification state the methods' pre- and post-conditions respectively. To be consistent with the Liskov and Wing paper and the Larch approach, pre-conditions are single-state predicates and post-conditions are two-state predicates.[1] In Larch interface specifications, as shown in Figure 3, the absence of a **requires** clause stands for the pre-condition "true". The *get_color* method returns the color of the (i, j)th array element of g. The *overlay* function defined in GIFImage returns the color value of the (i, j)th array element of the last frame in the sequence that gives a value for (i, j); otherwise, it returns BLACK, a distinguished color value, introduced in the LSL trait for colors. For example, if there are three frames in the frame sequence and for a given (i, j), the first frame maps the array element to BLACK, the second to RED, and the third does not map (i, j) to any color (because it is not within its bounds), then RED is returned.

To ensure that the specification is *consistent*, the specifier must show that each creator for the type τ establishes τ's invariant and each of τ's methods preserves the invariant. These are standard conditions and

[1]However, since we write formal specifications for only immutable types in this paper, to increase their readability, we take the liberty of not qualifying variables denoting objects with *pre* or *post* subscripts since $x_{pre} = x_{post}$ for all objects x.

their proofs are typically straightforward.

The inclusion of pre- and post-conditions in the specification of a type's methods allows us to relate the behaviors of two different types; 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., addition and exponentiation for integers) may have dramatically different behavior. Relying on just signature information identifies these methods that behave differently; thus, finer subtyping distinctions can be made when behavioral information is used in addition to signature information.

2.3 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, σ and τ. Since in general the value space for objects of type σ will be different from the value space for those of type τ we need to relate the different value spaces; we use an *abstraction function*, α, to define this relationship. Also since in general the names of the methods of type σ can be different from those of type τ we need to relate which method of σ corresponds to which method of τ; we use a *renaming* map, ν, to define this correspondence. (In a programming language like Java, this is just the identity map, as realized though method overloading.)

σ is a *subtype* of τ if the following two 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 $\alpha(s)$.
2. Subtype methods preserve the supertype methods' behavior. If m is a subtype method then let n be the corresponding $\nu(m)$ method of the supertype.
   - Signature rules
     - 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.
   - Methods rules
     - n's pre-condition implies m's (under the abstraction function).
     - m's post-condition implies n's (under the abstraction function).

The methods rules are completely analogous to the contra/covariant signature rules.

The formal definition of the subtype relation, $<$, is given in Figure 4. [2] It relates two types, σ and τ, each of whose specifications we assume respectively preserves its invariant, I_σ and I_τ. In the methods rules, since x is an object of type σ, its value (x_{pre} or x_{post}) is a member of S and therefore cannot be used directly in the predicates about τ objects (which are in terms of values in T). The abstraction function α is used to translate these values so that the predicates about τ objects make sense.

Why does this subtype relation guarantee that the No Surprises Requirement holds? Recall that the Requirement refers vaguely to "properties." This definition of subtype guarantees that certain properties of the supertype—those stated explicitly or provable from a type's specification—are preserved by the subtype. The first condition directly relates the invariant 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.

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.

Figure 5 gives a type specification for pixel_map. To show that GIF is a subtype of pixel_map (Figure 2, we define the following abstraction function:

[2] It is a simplification of Liskov and Wing's "constraint" rule definition, Fig. 4 of [9], by omission of the constraint rule condition.

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, $\alpha : S \to T$, and a renaming map, $\nu : M \to N$, such that:

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

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

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 a_i and that of m_σ is b_i, then $\forall i\ .\ a_i < b_i$.
     - *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 a and m_σ's be b. Then $b < a$.
     - *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.* $m_\tau.pre[\alpha(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre$.
     - *Post-condition rule.* $m_\sigma.post \Rightarrow m_\tau.post[\alpha(x_{pre})/x_{pre}, \alpha(x_{post})/x_{post}]$

Figure 4: Definition of the Subtype Relation

α_G^{PM} : G $\to$ PM
$\forall i, j$: Integer . $\alpha_G^{PM}(g)[i,j] = overlay(g, i, j)$

Using this abstraction function, the proofs that the invariant, signature, and methods rules hold are straightforward.

3 Respects

3.1 Definition of Respectful Type Converter

Suppose we have two types $A = \langle O_A, V_A, M_A\rangle$ and $B = \langle O_B, V_B, M_B\rangle$. A converter, C, is a partial function from V_A to V_B. Thus when we say that a converter maps from type A to type B we mean more precisely that it maps the value space of type A to the value space of type B; for notational convenience, we continue to write the signature of C as $A \to B$.

Let T be a type that is a common ancestor of A and B in a given type hierarchy. T is a supertype of both A and B. Then there exist ancestor types, $A_1 \ldots A_n$, between A and T such that there exist the following abstraction functions:

$\alpha_0 : A \to A_1$
...
$\alpha_i : A_i \to A_{i+1}$
...
$\alpha_n : A_n \to T$

Assuming for all $1 \leq i \leq n\ .\ dom(\alpha_i) \subseteq ran(\alpha_{i-1})$, let α be the functional composition of α_i:

$$\alpha = \alpha_n \circ \ldots \circ \alpha_0$$

pixel_map: **type**

uses PixelMap (pixel_map **for** PM)
for all p: pixel_map

invariant *true*

color *get_color* (i, j: int)
ensures $result = p[i, j]$

bool *is_set* (i, j: int)
ensures $result = (p[i, j] \neq \text{BLACK})$

bool *equal* (a: pixel_map)
ensures $result = (a = p)$

end pixel_map

Figure 5: A Larch Type Specification for Pixel Maps

For B we similarly define B_i, β_i, and β for $1 \leq i \leq m$. Figure 6 illustrates these constructs.

Figure 7 gives the definition of the *respects* relation for a converter $C : A \rightarrow B$ and type T. The first condition requires that m's pre-condition holds for a's abstracted value under α iff it holds for the converted value of a abstracted under β. Thus from T's viewpoint, if m is defined for A's values, it should be defined for B's values, and vice versa. The second condition requires that m's post-condition holds for a's abstracted value under α iff it holds for the converted value of a abstracted under β. Thus, given that m is defined, then its behavior must be the same for A's values and B's values from T's viewpoint.

Both conditions together guarantee that T's behavior is preserved by the conversion of objects of type A to those of type B. Informally, T cannot perceive a difference between the original A object and the converted B object; thus C *respects* T.

Finally, if a and $C(a)$ abstractly map to the same value in T (that is, $\alpha(a) = \beta(C(a))$) for all a in the domain of C, then the *respects* relation trivially follows. This special case is often useful in proofs that a converter respects a type, as we will see in the next section.

3.2 PNG and GIF Example Revisited

Let us look at the PNG to GIF example more carefully. First, we give the type specification for PNG images and abstraction functions that enable us to argue the remaining subtype relationships of Figure 2. Then we consider converters between PNG and GIF, to argue that no total converter respects pixel_map but that some respect bitmap.

The type specification for PNG images is given in Figure 8. PNG images differ from pixel_map objects in two ways: (1) they are framed and (2) associated with each PNG object, p, is a "gamma" value, denoted *gamma(p)*, used in a *gamma correction* function, gc. The gamma correction function corrects for differences among monitors; some are dimmer than others and thus have different color balances. We abstract from the intricacies of gamma correction functions; for our purposes here, they take as arguments a color, an input gamma factor, an output gamma factor, and return a color. The constant, STDG, is the standard gamma value for normal monitors. The Gammas and PNGImage traits are in Appendix A.

Note to show that PNG is a subtype of pixel_map we have a nontrivial application of the renaming map, ν, where $\nu(\mathit{get\_corrected\_color}) = \mathit{get\_color}$. (Only the last three methods for PNG have corresponding supertype methods; the rest are left unmapped by ν.)

We define below two abstraction functions: one to show that PNG is a subtype of pixel_map:

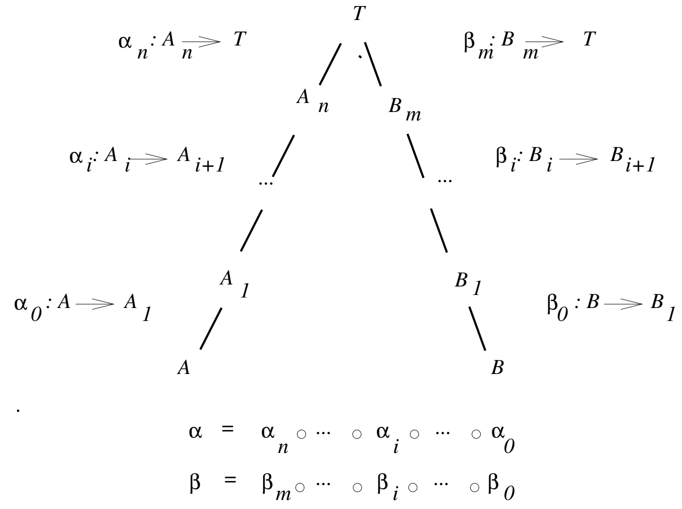

Figure 6: Two Compositions of Abstraction Functions

Definition of respects relation: Let $C : A \to B$ be a converter function, a partial function mapping values of type A to values of type B. Let T be an ancestor of both A and B in a given type hierarchy. Then *converter C respects T* if for each method m of T, $\forall a \in dom(C)$:

1. $m.pre_T[\alpha(a)/x_{pre}] \Leftrightarrow m.pre_T[\beta(C(a))/x_{pre}]$ and
2. $m.post_T[\alpha(a)/x_{pre}, \alpha(a)/x_{post}] \Leftrightarrow m.post_T[\beta(C(a))/x_{pre}, \beta(C(a))/x_{post}]$.

where α and β are defined in text and depicted in Figure 6.

Figure 7: Definition of the Respects Relation

$$\alpha_P^{PM} : \mathrm{P} \to \mathrm{BM}$$

$$\forall i, j : \text{Integer} \ . \ \alpha_P^{PM}(p)[i,j] = \begin{cases} gc(p[i,j], gamma(p), STDG) & \text{if } xmin(p) \leq i \leq xmax(p) \land \\ & \quad ymin(p) \leq j \leq ymax(p) \\ \text{BLACK} & \text{otherwise} \end{cases}$$

and one to show that pixel_map is a subtype of bitmap:

$$\alpha_{PM}^{BM} : \mathrm{PM} \to \mathrm{BM}$$

$$\forall i, j : \text{Integer} \ . \ \alpha_{PM}^{BM}(pm)[i,j] = \begin{cases} \text{set} & \text{if } pm[i,j] \neq \text{BLACK} \\ \text{clear} & \text{if } pm[i,j] = \text{BLACK} \end{cases}$$

Consider a converter, $C : \mathrm{P} \to \mathrm{G}$, that maps values of PNG images to GIF values.

Claim 1 *There is no such converter that respects pixel_map, if the converter is defined for PNG images of more than 256 colors.*

Proof: A simple counting argument suffices to prove this. Let p be the value of a PNG image, where $\mid colorrange(p) \mid = \ n$ *and* $n > 256$. *Then the corresponding pixel_map value* $\alpha_P^{PM}(P)$ *also has at least 256 colors. This holds because the color value of* $\alpha_P^{PM}(p) = gc(p[i,j], gamma(p), STDG)$. *So every array element of p maps to some array element of* $\alpha_P^{PM}(p)$. *Furthermore, if two array elements in p have different colors, so do the corresponding cells in* $\alpha_P^{PM}(p)$. *To prove this, we show that if two gamma corrected colors are the same, then the original colors,* c_1 *and* c_2, *also had to be the same, i.e.,*

Suppose

1. $gc(c_1, gamma(p), STDG) = gc(c_2, gamma(p), STDG)$

PNG: **type**

uses PNGImage (PNG **for** P)
for all p: PNG

invariant $true$

color *get_uncorrected_color*(*i, j:* int)
requires $xmin(p) \leq i \leq xmax(p) \wedge ymin(p) \leq j \leq ymax(p)$
ensures $result = p[i, j]$

gamma *get_gamma* ()
ensures $result = gamma(p)$

int *get_xmin* ()
ensures $result = xmin(p)$

... and similarly for *get_xmax*, *get_ymin*, and *get_ymax* ...

color *get_corrected_color* (*i, j:* int)
ensures $xmin(p) \leq i \leq xmax(p) \wedge ymin(p) \leq j \leq ymax(p)$
$\Rightarrow result = gc(p[i, j], gamma(p), \mathrm{STDG}) \ \wedge$
$xmin(p) \geq i \vee i \geq xmax(p) \vee ymin(p) \geq j \vee j \geq ymax(p)$
$\Rightarrow result = \mathrm{BLACK}$

bool *is_set* (*i, j:* int)
ensures $result = (gc(p[i, j], gamma(p), \mathrm{STDG}) \neq \mathrm{BLACK})$

bool *equal* (*a:* GIF)
ensures $result = (a = p)$

end PNG

Figure 8: A Larch Type Specification for PNG Images

By the "transitivity" and "reflexivity" properties of gamma correction functions (see Appendix A), we know that

2. $gc(gc(c_1, gamma(p), STDG), STDG, gamma(p)) = c_1$

By substitution in line 1, we get

3. $gc(gc(c_2, gamma(p), STDG), STDG, gamma(p)) = c_1$

Yielding

4. $c_2 = c_1$

So if there are $n > 256$ colors in p then there are also at least n in $\alpha_P^{PM}(p)$.

The conversion of p to a GIF image $C(p)$ can have a maximum of 256 colors, by the invariant of GIF image. Furthermore, there is no way the abstraction mapping of $C(p)$ to a pixel_map value, $\alpha_G^{PM}(C(p))$, can add any further colors (except for BLACK), since we see from the GIF to pixel_map abstraction function, α_G^{PM} that every element in the color set of $\alpha_G^{PM}(C(p))$ is either BLACK or one of the colors used in one of the frames of $C(p)$. Therefore, some of the colors available in $\alpha_P^{PM}(p)$ are not available in $\alpha_G^{PM}(C(p))$. So the value of get_color(i, j) for some i and j will be different. Therefore, the converter cannot respect pixel_map. □

It is possible, however, to have a converter from PNG images to GIF images that respects the bitmap type.

Claim 2 *There exist such converters that respect bitmap.*

Proof: By existence. Here is a simple-minded converter[3]*:*

$$\begin{aligned}
&C : P \rightarrow G\\
&C(p) = g \text{ where}\\
&\quad xmin(g) = xmin(p) \ \wedge xmax(g) = xmax(p) \ \wedge\\
&\quad ymin(g) = ymin(p) \ \wedge ymax(g) = ymax(p) \ \wedge\\
&\quad \forall i, j : \text{Integer} \ . \ g[i,j] = \begin{cases} BLACK & \text{if } gc(p[i,j], gamma(p), STDG) = BLACK \\ WHITE & \text{otherwise} \end{cases}
\end{aligned}$$

Composing the abstractions functions, $\alpha_{PM}^{BM} \circ \alpha_P^{PM}$, we get a bitmap, b, for a given PNG image p:

$$[\mathbf{1}] \quad \forall i, j : \text{Integer} \ . \ b[i,j] = \begin{cases} set & \text{if } xmin(p) \leq i \leq xmax(p) \wedge ymin(p) \leq j \leq ymax(p) \ \wedge \\ & \quad gc(p[i,j], gamma(p), STDG) \neq BLACK \\ clear & \text{otherwise} \end{cases}$$

Composing the abstractions functions, $\alpha_{PM}^{BM} \circ \alpha_G^{PM}$, we get a bitmap, b, for a given GIF image $C(p)$:

$$[\mathbf{2}] \quad \forall i, j : \text{Integer} \ . \ b[i,j] = \begin{cases} set & overlay(C(p)[i,j]) \neq BLACK \\ clear & \text{otherwise} \end{cases}$$

By substituting the definition of overlay*, given the definition of C above, we get that* **[2]** = **[1]**. *Since the two bitmaps are the same, the converter C respects bitmap.* □

4 An Application: The TOM Conversion Service

4.1 Overview of TOM

As part of his Ph.D. thesis, Ockerbloom (the second author) invented a Typed Object Model [10], a data model involving objects, types, and their associated metadata. The thesis includes definitions and explanations of *intersubstitutability*, which is like our *respects* relation, though formulated differently. The thesis also includes full information about the design of TOM, and describes experience building and using TOM-based applications.

[3] A more realistic converter would not map all non-BLACK colors to WHITE, but to something closer to the original color; however, the above simplifies the proof.

Ockerbloom implemented an instance of the TOM model, a type broker that allows users in a distributed environment to store types and type conversion functions, to register new ones, and to find existing ones. The kinds of 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 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*.

Users can compose available converters to produce an object of a desired target type. For example, to make a fourteen-year old Scribe document available on the Web, the first author used a *scribe2latex* converter to produce a LaTeX file and then an enhanced *latex2html* converter to produce the Web version. This scenario is similar to that described in the introduction (Section 1.1): here, the composition of the two converters preserves the semantics of the original Scribe document. Ironically, the LaTeX program failed to run successfully on the intermediate LaTeX file, but for this application, it did not matter; it was the end result—the HTML file—that mattered.

Some of TOM's converters, especially the home-grown ones, are defined in terms of others. For example, 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 steps include converting postscript files to ppm files, resizing and rotating ppm files, and converting ppm files to GIF files. TOM users of the *powerpoint2html* converter see none of these intermediate steps.

We discuss TOM's usage and status in more detail in Section 7.

4.2 Snippets from the TOM Type Hierarchy

When Ockerbloom 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.

Despite this design simplification of disallowing mutable types, TOM does support an interesting subtype hierarchy. Figure 9 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, containing 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 so 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.

We carefully designed the TOM 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 between types in the TOM hierarchy is relatively easy. There are two main cases:

Case 1. If no changes to old methods are made then the proof is trivial. 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 extra state information, e.g., more fields in a record, more instance variables of an object, or extra attributes. The abstraction function α is the obvious many-to-one function that throws away the extra state information. The behavior of the subtype methods is identical to that of the corresponding supertype methods and new methods (defined on the extra state) have no corresponding supertype method.

For example, bordered windows, colored windows, and scrollable windows are all extended subtypes of

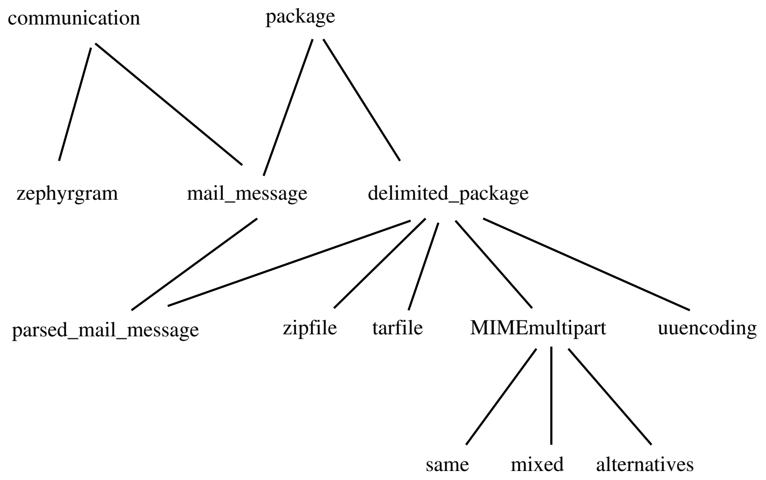

Figure 9: Part of TOM's Type Hierarchy

a more generic window type [5]. In their paper, Liskov and Wing call these *extension* subtypes since the subtype extends the supertype by providing additional state and correspondingly additional methods.

Case 2. 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 are preserved (given that Part 2 holds and that the specifications are consistent). The abstraction function is usually into rather than onto.

For example, in Figure 2 GIF image is a subtype of pixel_map by constraining the colors over which a GIF image element can range. In their paper, Liskov and Wing call these *constrained subtypes* because the degree of variability is reduced in the subtype.

In general, a subtype might both extend and constrain its supertype. For example, in Figure 9 uuencoding constrains delimited_package by constraining the number of items in the package to be exactly one; it extends delimited_package by including UNIX file protection bits as additional state.

Consider another snippet of the TOM type hierarchy that deals with document types shown in Figure 10. Suppose that LaTeX and HTML documents both abstractly map to a hierarchical document type that specifies a hierarchy of sections and the text included in each section. The converter function *latex2html* respects the hierarchical document type if it preserves the body text and the section hierarchy implied by LaTeX sectioning commands and HTML header levels. A looser conversion between the same two types might preserve the body text, but not the section hierarchy, and thus respects the generic document type but not the hierarchical document type. Note that type conversions may involve not only siblings, but cousins, or ancestors, as well. Hence, an *html2ascii* converter might simply respect the generic document type. A converter from LaTeX_with_URLs to basic LaTeX, defined so that it follows the abstraction function between the same types, respects the basic LaTeXtype, i.e., the parent type.

5 Incorporating Concrete Types

So far we have discussed the *respects* relation in terms of abstract types since the subtype relation is defined in terms of a relation on abstract types. It makes sense, however, to consider the *respects* relation in terms of

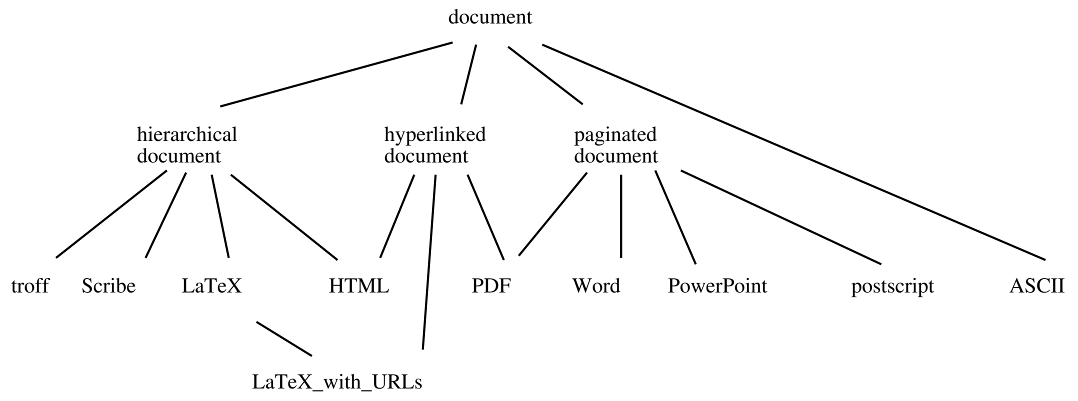

Figure 10: Another Part of TOM's Type Hierarchy

concrete types too. For example, when we implement an abstract type in a programming language, we choose a representation type for the abstract type and define the abstract type's methods in terms of methods on the representation type.

The TOM context introduces another kind of concrete type. When a user retrieves an object from a remote site, in reality that object is encoded in terms of some transmissible type, a concrete representation of the abstract object. These transmissible types are in turn represented in terms of primitive types that the underlying communication substrate understands; for TOM, and for the purposes of this paper, it suffices that every transmissible object be representable in terms of sequences of bytes; part of the byte sequence might represent metadata (e.g., the name of the abstract type) and the rest represents the data object itself.

Both of these kinds of concrete types may give rise to a new kind of converter, that from a concrete type to another concrete type. For example, in programming languages, if we have an abstract point type with two different representations, one using Cartesian coordinates and one using polar, we may want to implement a converter that takes any Cartesian point and produces the corresponding polar coordinates. Similarly, for an abstract matrix type, we may want to represent matrices in terms of both row-order and column-order and define converters between the two.

In the TOM context, integers may be represented in terms of a 32-bit sign extension byte sequence or a twos-complement, little-endian byte sequence, or even ASCII strings. These are all plausible concrete representations of integers and conversions between them should respect the abstract integer type.

5.1 Extending the Definition of Respects

To extend our definition of *respects* to accommodate converters from concrete type to concrete type, we borrow from the programming language community: we use the very same abstraction function used to prove the correctness of data representations first introduced by Hoare in 1974 [6].

Let converter $C : A_{conc} \rightarrow B_{conc}$ be defined on two concrete types A_{conc} and B_{conc}. Then if A_{conc} and B_{conc} are correct implementations of (abstract) types A and B, respectively, there exist abstraction functions:

$$\mathbf{A} : A_{conc} \rightarrow A$$
$$\mathbf{B} : B_{conc} \rightarrow B$$

We modify the definition of *respects* by modifying the definitions of α and β of Section 3.1 accordingly. Assuming that $dom(\alpha_0) \subseteq ran(\mathbf{A})$ and $dom(\beta_0) \subseteq ran(\mathbf{B})$, we define

$$\alpha = \alpha_n \circ \ldots \circ \alpha_0 \circ \mathbf{A}$$
$$\beta = \beta_m \circ \ldots \circ \beta_0 \circ \mathbf{B}$$

That is, we first apply the abstraction function **A** on the concrete value of type A_{conc} to form an abstract value of type A; we do the same to the concrete value of type B_{conc} using **B**. C respects T if the same condition holds as before, but using the revised α and β abstraction functions defined above. In the case that the converter C maps an abstract type to a concrete one or vice versa, then we can omit the application of **A** or **B** as appropriate.

5.2 Examples

A degenerate example of this extension to the respects relation is the abstract point example. If we write a conversion function from Cartesian coordinates to polar coordinates, abstractly they are the same point and should exhibit the same behavior as defined by the specification of the abstract point type.

A less degenerate example from the TOM application is in representing directories. A directory type might be represented as a list of strings type. The list of strings type might in turn be represented by the the sequence of bytes type, i.e., a transmissible type. A client who wishes to view the contents of a remote directory can do so even though the client's file system (e.g., Window NT) may differ from the server's (e.g., afs).

A common situation in using TOM involves a client retrieving an object from a remote server. The client would like to view the object, originally in type A, as an object of type B so far as it respects some type T. In the client's mind some abstract conversion from A to B is being performed. The problem is that the A object has to be represented in terms of something transmissible across the wire; so first it is encoded into some transmissible type and then the client decodes the transmitted object into a B object. Here A_{conc} and B_{conc} may very well be the same (transmissible) type, e.g., sequence of bytes, but at both the client and server sides abstract interpretations are defined on A_{conc} and B_{conc} to yield respectively objects of A and B. For example, suppose a client fetches a compressed Word file from a Web server and wants to view it as an HTML file in a Web browser. Both the Word file and the HTML file are ultimately represented as sequences of bytes. The conversions required to uncompress and to convert the Word document to HTML respect the generic document type, via the abstraction that captures both their relations to the generic document types, and the relation of their byte-sequence representation to their abstract types Word and HTML.

Finally, consider the problem of parsing integers retrieved from a text document and storing them in a packed byte array. We can model this problem as a conversion from a string-based representation of integers to a byte-based representation. We have an abstract type int, a concrete type ibytes, and another concrete type istr. The abstraction function α: istr $\rightarrow$ int is *atoi*, i.e., the standard C library function. The abstraction function β: ibytes $\rightarrow$ int is defined as follows for a given ibytes value b:

$$b[0] + 256 * b[1] + ...256^{n-1} * b[n-1]$$

where n is the number of bytes in b. Then $\beta^{-1} \circ \alpha$ is a conversion from istr to ibytes that respects int. This example illustrates a common way to find a conversion function: In general, if types A and B are "below" type T by a composition of subtype and representation relations, and α is the abstraction function from A to T and β is the abstraction function from B to T, and β is invertible, then $\beta^{-1} \circ \alpha$ is a conversion from A to B that respects T.

6 Two Other Applications of the Respects Relation

The notion of a converter respecting a type shows up in other contexts besides explicit type conversion, provided by the TOM Conversion Server. We cast one critical aspect of type evolution in terms of our respects relationship and we show how respectful type converters enable reuse in the context of interoperability.

6.1 Type Evolution

In Clamen's survey on type evolution he states that object-oriented databases tend to rely on two adaptation strategies for handling the evolution of database schemas (which can be modeled as types) [3]. The first strategy converts instances from old types to new ones. The second strategy uses emulation to allow instances of old types to be used through interfaces of both the new types and the old type. Respectful type converters play a role in both strategies. For conversions, they ensure that appropriate information is preserved. They aid in emulation, by converting between types when an object of an old type is used through a new type's interface, in a way that ensures that the new type's methods, when called on the converted object, are consistent with the old type's methods called on the original object. (We will see an example of this strategy in the next section with our *zipfile2tarfile* converter.)

The type evolution problem arises most commonly in the context of databases. A change to a database schema, e.g., the addition of a new field in a record, necessitates changing all records consistent with the old schema to be consistent with the new one. For example, suppose we maintain an address book of colleagues; each entry type has a name, address, office phone number, home phone number, fax number, and e-mail address. We wish to update our address book so that each entry can contain a url also. This is a simple, but common, example of type evolution.

As we have seen, a simple change like adding new state information to a type is just a simple application of subtyping. Here, the new address book entry type is a subtype of the old one. As before, the abstraction function loses the extra information defined by the subtype.

Where do converters play a role? It answers in part the question of what the relationship between the orginal and evolved type is. In our example, the converter C : entry $\rightarrow$ entry_and_url respects the entry type. Here A and T of our general definition are the same type, entry. Intuitively, we want this converter to respect the original type because we want old code that used to work with old address book entries to continue to work with the new ones, even though new ones contain more state information.

Consider this less trivial example of type evolution where a type converter may or may not respect a given type, depending on the type converter's definition. Suppose we have a type, HMS, that keeps track of time in terms of hours, minutes, and seconds. HMS has a subtype, HMSZ, that also includes the time zone. The abstraction function is again the obvious one that loses the time zone information. Consider two different type converters. The first, $C1$: HMS $\rightarrow$ HMSZ, adds "Eastern Standard Time" to all HMS values. This converter respects HMS. The second, $C2$: HMS $\rightarrow$ HMSZ, converts all HMS values to times according to Greenwich Mean Time and then adds "Greenwich Mean Time" to produce an HMSZ value. $C2$ does not respect HMS since for any non-GMT time value the hours of the HMS and HMSZ values would be different.

Thus the respects relation helps characterize what information should be preserved when types evolve. We should not be able to evolve types without any thought as to what the relationship between the original and evolved type should be. In other words, we have an instance of the same question as for type converters, and our answer is similar: type evolution should preserve some information between the old and new types. Our *respects* relation gives type evolution a correctness condition to satisfy because it is defined in terms of a type, not just the evolution (conversion) function.

6.2 Reuse for Interoperability

At the heart of the interoperability problem [1] is resolving mismatches among heterogeneous components. For n components, e.g., n different types, at worst we need n^2 converters to do pair-wise communication between the components to achieve "point-to-point connectivity." We can reduce the number by requiring the use of a common language or standard, thereby requiring only $2*n$ translation functions (n encode and n decode functions) for full connectivity. Regardless, interoperability requires some number of type converters.

If we know that the types are structured in way that preserves some information, then we can reduce the number of converters we need to implement by exploiting common ancestors for a group of types.

Suppose type T, a virtual type, defines five methods and has two subtypes, A and B. Suppose also there is a converter from A to B that respects T. Then instead of implementing those five methods twice, we need only implement each once and rely on the single converter, to effect their behavior on A objects. Thus, we implement six functions (five methods plus one converter) rather than ten. We not only save some implementation effort but we *reuse* the effort expended already for one type.

In general for n heterogeneous components, each of which is to have the same m methods defined, if all n components are subtypes of some type T, then we need only implement $m + (n - 1)$ functions rather than $m * n$ functions. The more subtypes of T and the more methods "shared" by those subtypes, the greater the potential reuse.

In practice, there may exist off-the-shelf converters, e.g., commercial products, between heterogeneous components; moreover, these components are unlikely to be arranged in a type hierarchy. To apply our ideas in practice, we typically identify some common supertype, even if it means defining a virtual supertype [9], T, such that the use of these existing converters guarantee that T's behavior is preserved. Thus we can provide stronger, formalizable guarantees to users of these off-the-shelf converters.

Our TOM type hierarchy illustrates a real case of this scenario. There already exist *zipfile2tarfile* and *tarfile2zipfile* type converters. The package supertype of zipfile and tarfile (and the three others shown in Figure 9) is actually a virtual type that exports over one-half dozen methods, all common to its subtypes. These methods include getting the ith item from the package, returning the count of items in the package, unwrapping the ith item, getting the name of the ith item in the package, and returning a list of names of items in the order they appear in the package. Rather than implement all the methods for both zipfile and tarfile, we implemented them for only tarfile. Note that we rely on the *respects* relation for it to be meaningful to do the conversion on zipfiles, treating them as tarfiles.

7 TOM Status

7.1 TOM Use

The Web site for the TOM Conversion Service at Carnegie Mellon is: `http://tom.cs.cmu.edu/`. It supports roughly 100 abstract data types, a few hundred concrete data types, and over 300 type converters (including over 200 meaningful compositions of about 70 primitive converters). By July 1997, the number of accesses to the TOM conversion service stabilized to between 2100 and 2600 per month, which is an average of between 70 and 85 per day. Accesses came from over 200 sites in over 20 countries in six continents from all types of organizations including educational, government, and commercial institutions. One class of the most popular converters are those to unwrap mail messages with embedded files. Another class consists of those which take a document of one type and produce a file of a different type, typically for viewing or printing. The most common source types for converting are mail messages, Microsoft Word, PowerPoint, postscript, LaTeX, and pdf files; the most common target types are Web sections, HTML, text, postscript, and GIF images.

The TOM Conversion Service includes a Web-based user interface that hides much of the complexity of type conversion from the user, in three important ways:

- TOM is a connected network of *type brokers*. If a user makes a request to one instance of a TOM broker, 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. If the user gives TOM a source type and a target type, 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. TOM can formulate conversion plans that respect types, since if conversion C_1 respects type T_1 and conversion C_2 respects type T_2, then $C_1 \circ C_2$ respects any common supertype of both T_1 and T_2.

- 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.

7.2 Revisiting TOM State

Recall the model of state introduced in Section 2.1. A *state* consists of an *environment*, which maps variables to objects, and a *store*, which maps objects to values.

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

TOM supports *handles* to objects; file names and URLs are two subtypes of the handle type. Handles are also TOM immutable objects and provide a *dereference* method. For example, the content of a file is a TOM object, not the file itself; a file name is a TOM object, which when dereferenced, refers to the content of a file. These handles make up exactly the set *Var*, the domain of the environment.

By definition, values of immutable objects cannot change and since TOM's objects are all immutable the store of any TOM state never changes (except by the addition of new objects). What about the environment, which keeps track of how to access objects? It turns out that the environment may change. In particular, for a given state, $\rho = \langle e, s \rangle$, while its store component, $\rho.s$, cannot change, its environment component, $\rho.e$, may change.

More precisely, the *binding* between handles to TOM objects may change and these changes are effected by users of TOM. More importantly, *TOM has no control over these changes.* In the context of programming languages, when a program is run, the program's environment is the machine it runs on; the implementation of the programming language's runtime system has complete control over the program's state, including its environment. Thus, a new variable declared can be given an initial binding to an object; this binding can be added to the program's environment. When the program terminates, all these bindings presumably disappear. The program and its environment are self-contained.

TOM operates in an open distributed environment where users can change the bindings between handles and objects. TOM has no control over this mapping. The consequence is that 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. Futhermore, two different handles, e.g., two different URLs, may dereference to the same file contents. TOM cannot control or might not 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 appear to users as 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: the potential mutability of a system's environment must be taken into consideration when accessing the repository's objects. What may be immutable in one context may *appear* to be mutable from a broader perspective.

8 Summary and Future Work

In this paper, we defined a novel notion of *respectful type converters* to capture what information a conversion function preserves when transforming objects of one type to another. We greatly leverage off the Liskov and Wing notion of behavioral subtyping to characterize this information succinctly. Their framework gives us the key technical tool we need, in particular, the abstraction functions, α_i, to define formally how two different objects can be viewed as the same. We also extended our definition to incorporate concrete types by leveraging off Hoare's abstraction functions for proving the correctness of the implementations of abstract types.

An alternative, more general, approach to defining *respects* would neither require that types A, B, and T be related in a type hierarchy at all nor require that types A_{conc} and B_{conc} be related to A and B in any implementation hierarchy. We, however, would still need to assume the existence of some functions that map values of one type to another; these functions would serve the purpose of our abstraction functions, but perhaps be otherwise unconstrained.

We also described the utility of respectful type converters in the context of working system: the Carnegie Mellon TOM Conversion Service. We briefly suggested the relevance of the respects relation for type evolution and interoperability. From our application examples, we can characterize the three most common kinds of converters as

1. Those mapping from one abstract type to another abstract type. Here the converter respects a third abstract type, usually a virtual type introduced in the type hierarchy. Examples include conversions between different document types of Section 4.2.

2. Those mapping from a supertype to a subtype or vice versa; the converter respects the supertype, e.g., the time zone example of Section 5.

3. Those mapping from a concrete type to a concrete type; the converter respects the abstract type that both concrete types implement, e.g., the point, matrix, and integer examples of Section 5.

By sticking to the design decision that no TOM object can be mutable, we 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 simple and usually straightforward.

The real power of the Liskov and Wing definition of subtype, however, is in handling mutable types. Though we do not exploit this power in this paper for simplicity and because our application did not demand it, our definition of *respects* can be extended to accommodate mutable types. Type specifications would include a predicate, called *type constraints* by Liskov and Wing, which captures what behavior may *not* change from state to state; hence it captures additional "invariant" properties of mutable objects. The definition of *respects* would additionally need to consider the preservation of type constraints and the behavior of mutator methods.

References

[1] Frank Bamberger, Peter Ford, and Jeannette M. Wing. *Interoperability*, chapter C.8, pages 67–71. Interuniversity Communications Council, Inc. (EDUCOM), 1994.

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

[3] Stewart Clamen. Type evolution and instance adaptation. Technical Report CMU-CS-92-133R, Carnegie Mellon Computer Science Department, Pittsburgh, PA, June 1992.

[4] 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.

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

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

[7] 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.

[8] Barbara Liskov. Data abstraction and hierarchy. In *OOPSLA'87: Addendum to the Proceedings*, 1987.

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

[10] John Ockerbloom. Mediating among diverse data formats. Technical Report CMU-CS-98-102, Carnegie Mellon Computer Science Department, Pittsburgh, PA, January 1998. Ph.D. thesis.

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

Appendix A: Larch Traits and Type Specifications

This appendix contains the following Larch specifications: bitmap interface type specification, BitMap trait, Color trait for color literals, ColorSet trait for sets of colors, Frame trait, FrameSeq trait, GIFImage trait, Gammas trait, PNGImage trait, and PixelMap trait. Appendix A of the Larch Book [7] contains traits for Boolean, Integer, FloatingPoing, Set, Deque, and Array2, all of which we use below.

bitmap: **type**

uses BitMap (bitmap **for** BM)
for all b: bitmap

  invariant true

  bool *is_set* (*i, j:* int)
    ensures result = $(b[i, j] = set)$

  bool *equal* (*a:* bitmap)
    ensures result = $(a = b)$

end bitmap

BitMap: **trait**
  includes Array2 (Bit, Integer, Integer, BM), Bits
end BitMap

Bits: **trait**
  includes Boolean (Bit **for** Bool, set **for** true, clear **for** false)
end Bits

ColorLiterals: **trait**
  % A trait for N colors where BLACK = 0 and WHITE = 1 and N >> 256.
    Color **enumeration of** BLACK, WHITE, 2, ..., N-1
end ColorLiterals

ColorSet(Color, CS): **trait**
  includes ColorLiterals, Set (Color, CS)
end ColorSet

Frame(F): **trait**
  includes Array2 (Color, Integer, Integer, F), ColorSet (Color, CS)
  introduces
    $xmin, xmax, ymin, ymax$: F $\rightarrow$ Integer
    $colorrange$: F $\rightarrow$ CS
  asserts for all i, j : Integer, f : F
    $xmin(f) \leq xmax(f)$
    $ymin(f) \leq ymax(f)$
    $(xmin(f) \leq i \leq xmax(f) \wedge ymin(f) \leq j \leq ymax(f)) \Rightarrow f[i, j] \in colorrange(f)$
end Frame

FrameSeq(F, FS): **trait**
  includes Deque (Frame, FS)
  introduces
    overlay: FS, Integer, Integer $\rightarrow$ Color

colorrange: FS $\rightarrow$ CS
__[__]: FS, Integer $\rightarrow$ F
asserts for all *i, j:* Integer, *f:* F, *fs:* FS
overlay(fs, i, j) = **if** *len(fs)* = 0 **then** BLACK **else**
if $xmin(last(fs)) \leq i \leq xmax(last(fs)) \wedge ymin(last(fs)) \leq j \leq ymax(last(fs))$
then $last(fs)[i, j]$
else $overlay(init(fs), i, j)$
$colorrange(empty) = \{\}$
$colorrange(fs \vdash f)) = colorrange \cup colorange(f)$
fs[*i*] = **if** *i* = *len(fs)* **then** *last(fs)* **else** *fs*[*i-1*]
exempting
$\forall i : Integer \ . \ empty[i]$
$\forall i \leq 0 \ . \ fs[i]$
$\forall i \geq len(fs) \ . \ fs[i]$
end FrameSeq

GIFImage: **trait**
includes FrameSeq (G **for** FS), ColorSet(Color, CS)
asserts for all *g:* G
BLACK $\in colorrange(g)$
end GIFImage

Gammas: **trait**
includes FloatingPoint (Gamma **for** F)
introduces
STDG: $\rightarrow$ Gamma
gc : Color, Gamma, Gamma $\rightarrow$ Color
asserts for all *c*: Color, *g, h, i*: Gamma
$gc(c, g, g) = c$ "reflexivity"
$gc(gc(c, g, h), h, i) = gc(c, g, i)$ "transitivity"
end Gammas

PNGImage: **trait**
includes Frame (P **for** F), Gammas
introduces
gamma: P $\rightarrow$ Gamma
end PNGImage

PixelMap: **trait**
includes Array2 (Color, Integer, Integer, PM), ColorSet (Color, CS)
introduces
colorrange: PM $\rightarrow$ CS
asserts for all *i, j:* Integer, *pm:* PM
BLACK $\in colorrange(pm)$
$pm[i, j] \in colorrange(pm)$
end PixelMap