8

Respectful Type Converters for Mutable Types

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

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 $K : A \rightarrow B$ *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.

In previous work we defined *respects* for immutable types A, B, and T; in this chapter we extend our notion to handle conversions between mutable types. This extension is nontrivial since we need to consider an object's behavior as it varies over time. We present in detail two examples to illustrate our ideas: one for converting between PNG images and GIF images and another for converting between different kinds of bounded event queues. This work was inspired in building at Carnegie Mellon the Typed Object Model (TOM) conversion service, in daily use worldwide.

8.1 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 display, edit, 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, for example, 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.

8.1.1 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, 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 chapter 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* $K : A \rightarrow B$ *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 (through appropriate abstractions) 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, as defined by the type's interface specification. This chapter formalizes the 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 [LW94] 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 K respects type T, a supertype of both A and B, then T captures the behavioral information preserved by K.

In our previous work, "Respectful Type Converters" [WO99a] we defined the *respects* relation for conversions between immutable types only. (It was correspondingly based on a simplified version of Liskov and Wing's definition of subtype.) In this chapter, we present an enhanced version of our respects relation that captures

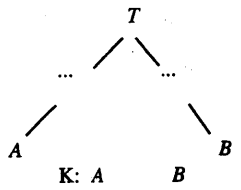

Fig. 8.1. Does Converter K Respect Type T?

important properties of conversions between mutable types. Specifically, for mutable types we say that a conversion *respects* a certain type T if an object with the converted value cannot be distinguished (using T's interface specification) from an object with the original value either at the time of conversion, or by analyzing any future computation on the object. That is, the future subhistory of the new object will not be inconsistent with the expectations raised by the past subhistory of the original object, given the constraints of type T.

Here is an example of why for some common ancestors, T, of A and B there may not exist respectful type converters from A to B (Figure 8.1). Consider a type family for images, depicted in Figure 8.2. Suppose that the PNG image and GIF image types are both subtypes of a pixel_map type 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 does not have a fixed 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. 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 nonblack color to black (or black to nonblack), 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.

8.1.2 Typed Object Model Context

At Carnegie Mellon we built a type broker, an instance of Ockerbloom's Typed Object Model (TOM) [Ock98], that provides a type conversion service. Our TOM type broker allows users in a distributed environment to store types and type conversion functions, to register new ones, and to find existing ones. It supports roughly 100

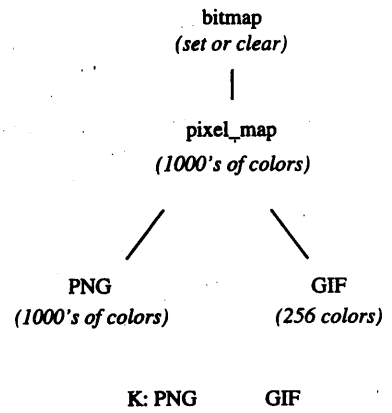

Fig. 8.2. A PNG to GIF converter that does not respect pixel_map might still respect bitmap. Conversely, it is easy to define a GIF to PNG converter that respects both pixel_map and bitmap.

abstract data types, a few hundred concrete data types, and over 300 type converters (including over 200 meaningful compositions of about 70 primitive converters). The kinds of types TOM supports today are different kinds of document types (for example, Word, LaTeX, PowerPoint, binhex, HTML) and "packages" of such document types (for example, 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* (that is, AdobeDistiller^{TM}), off-the-Web ones like *latex2html*, and some home-grown ones like *powerpoint2html*.

The website for Carnegie Mellon's TOM service is: `http://tom.cs.cmu.edu/`. As of May 1999, the number of accesses to the TOM conversion service stabilized to 5000 per month, which is an average of 167 per day. Accesses came from over 1000 sites in over 35 countries in six continents from all types of organizations including educational, government, and commercial institutions.

In designing the type hierarchy for this service, Ockerbloom made a deliberate decision to use only immutable types [Ock98], and hence our original paper [WO99a] on the *respects* relation ignored the complexities of mutability. However, Wing observed that though the objects are immutable, in their context of use—a distributed environment—the same problems of aliasing of mutable objects can arise [Win97]. For example, clients can access the same object through more than one naming scheme, for example, a URL, and a local file name; since objects are not necessarily uniquely named, this aliasing can lead to conflicting updates. Hence, in this chapter we investigate what *respects* means more generally, that is, in the presence of shared mutable objects.

Though our idea of respectful type converters was inspired by our use of TOM

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; payroll processing centers routinely need to convert large databases of employee records whenever extra fields are added to the relevant database schema; and of course, the infamous Year 2000 (Y2K) conversion problem is costing billions of dollars to fix [Jon98].

8.1.3 Roadmap to Rest of Paper

In this chapter we formally characterize the notion of when a converter *respects* a type. We first review in Section 8.2 background details leading up to the definition of Liskov and Wing's behavioral notion of subtyping [LW94]; this section is included to make this chapter self-contained. In Section 8.3 we exploit this notion of subtyping to define the *respects* relation between a converter and a type. In Section 8.4 we discuss in detail two examples to illustrate converters that do and do not respect types: one example of a type family for PNG and GIF images, and one for bounded event queues. We close with a discussion of related work and summary remarks.

8.2 Behavioral Subtyping

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

$$x\colon T := E$$

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 of an object of a type T that users rely on should hold even if the object is actually a member of a subtype S of T.

which guarantees Liskov's *substitutability* principle of subtypes [Lis87]. In their 1994 TOPLAS paper "A Behavioral Notion of Subtyping" Liskov and Wing [LW94] formalized this requirement in their definition of subtyping. The novel aspect of their subtype definition is the ability to handle mutable types, and in particular, a

type's *history* properties. Their specification logic limits the expressibility of history properties to be monotonic. For example, they can state that the bound of a queue stays the same but cannot state that it cyclically increases and decreases. Another example of a history property is that the value of an integer counter monotonically increases. They cannot state any liveness properties. We assume the same limitations herein.

Chapter 6 of this volume contains an extensive overview of many definitions of behavioral subtyping, including Liskov and Wing's. To provide background for our definition of *respects*, we first describe their model of objects and types, how they specify types, and then how they define the subtype relation. These definitions are all taken from the Liskov and Wing paper [LW94].

8.2.1 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.

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

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

We model a type as a triple, $\langle O, V, M \rangle$, where $O \subseteq Obj$ is a set of objects, $V \subseteq Val$ is a set of values, and M is a set of methods. Each method for an object is a *constructor*, an *observer*, or a *mutator*. Constructors of type τ return new objects of type τ; observers return results of other types; mutators modify the values of objects of type τ. A new object is one which does not exist in the domain of the state upon method invocation. An object is *immutable* if its value cannot change and otherwise it is *mutable*. A type is immutable if all of its objects are; otherwise it is mutable. We allow *mixed methods* where a constructor or an observer can also be a mutator. We also allow methods to signal exceptions; we assume termination exceptions, that is, 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*, that is, a program execution, is a sequence of alternating states and transitions starting in some initial state, ρ_0:

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

Each transition, Tr_i, of a computation sequence is a partial function on states. A *history* is the subsequence of states of a computation.

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

8.2.2 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, that is, the types of its arguments (in order), result, and signaled exceptions.
  - Its behavior in terms of preconditions and postconditions.
- A description of the type's history properties.

Figure 8.3 gives an example of a type specification for GIF images. We give formal specifications, written in the style of Larch [HGJ+93], 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 [GG89], to show that a subtype relation holds [Zar96].

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. The Appendix contains all traits used in all examples in this chapter, and here in particular, the GIFImage trait and those for frame sequences, frames, colors, and so forth.

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 for all states ρ in c:

GIF: **type**

uses GIFImage (gif **for** G)
for all g: GIF

invariant $| \mathit{colorrange}(g_\rho) | \leq 256$
constraint *true*

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

bool *set_color* (*i, j:* int; *c:* color)
modifies g
ensures **if** $| \mathit{colorrange}(\mathit{changepixel}(g_{pre}, i, j, c)) | \leq 256$
then $c \in \mathit{colorrange}(g_{post}) \wedge g_{post} = \mathit{changepixel}(g_{pre}, i, j, c) \wedge$
$\mathit{result} = \mathit{true}$
else $g_{pre} = g_{post} \wedge \mathit{result} = \mathit{false}$

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

end GIF

Fig. 8.3. A Larch Type Specification for GIF Images.

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

Whereas an invariant property is a property true of all states of an object, a history property is a property that is true of all sequences of states that result from any computation on that object. A type constraint in a Larch interface specification defines the history properties of the type's objects. The two-state predicate $\phi(x_{\rho_i}, x_{\rho_k})$ appearing in a **constraint** clause for type τ stands for the predicate: For all computations c, and for all states ρ_i and ρ_k in c such that $i < k$:

$$\forall x : \tau \; . \; x \in \mathit{dom}(\rho_i) \Rightarrow \phi(x_{\rho_i}, x_{\rho_k})$$

Note that we do not require that ρ_k be the immediate successor of ρ_i in c. The GIF example has the trivial constraint *true*. In Section 8.4.2 we will give examples of types with nontrivial constraints.

The **requires** and **ensures** clauses in the Larch interface specification state the methods' pre- and postconditions respectively. To be consistent with the Liskov and Wing paper and the Larch approach, preconditions are single-state predicates and postconditions are two-state predicates. The **modifies** clause states that the values of any objects it does *not* list do not change; the values of those listed may possibly change. The absence of a **requires** clause stands for the precondition *true*.

The absence of a **modifies** clause means that the method cannot change the values of any objects.

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 ColorLiterals trait. 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. The *set_color* method modifies the GIF object g by changing the final color of pixel (i, j) to c, and returns true, if the change would not make the resulting GIF have more than 256 colors. Otherwise it leaves the GIF object unchanged and returns false. The *get_frame* method returns the ith frame of the GIF object's value.

To ensure that the specification is *consistent*, the specifier must show that each creator for the type τ establishes τ's invariant, and that each of τ's methods both preserves the invariant and satisfies the constraint. These are standard conditions and their proofs are typically straightforward [LW94].

8.2.3 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; to define this correspondence we use a *renaming* map, ν, that maps names of methods of σ to names of methods of τ.

The formal definition of the subtype relation, $\preceq$, is given in Figure 8.4 †. It relates two types, σ and τ, each of whose specifications we assume are consistent. 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.

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. The third condition relates the overall histories of objects, guaranteeing that the possible

† It is Liskov and Wing's "constraint"-based subtype definition, and is taken from Fig. 4 of [LW94].

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

(i) Subtype invariants ensure supertype invariants.
- *Invariant Rule.* For all computations, c, and all states ρ in c for all $x : \sigma$:
  $I_\sigma \Rightarrow I_\tau[A(x_\rho)/x_\rho]$

(ii) 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 i argument types of m_τ is a and that of m_σ is b, then $\forall i \,.\, a_i \preceq b_i$.
  - *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 $b \preceq a$.
  - *Exception rule.* The exceptions signaled by m_σ are contained in the set of exceptions signaled by m_τ.
- *Methods rule.* For all $x : \sigma$:
  - *precondition rule.* $m_\tau.pre[\alpha(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre$.
  - *postcondition rule.*
    $m_\sigma.post \Rightarrow m_\tau.post[\alpha(x_{pre})/x_{pre}, \alpha(x_{post})/x_{post}]$

(iii) Subtype constraints ensure supertype constraints.
- *Constraint rule.* For all computations c, and all states ρ_i and ρ_k in c where $i < k$, for all $x : \sigma$:
  $C_\sigma \Rightarrow C_\tau[\alpha(x_{\rho_i})/x_{\rho_i}, \alpha(x_{\rho_k})/x_{\rho_k}]$

Fig. 8.4. Definition of the Subtype Relation.

histories (viewed abstractly) in the subtype specification are also possible histories in the supertype specification.

Figure 8.5 gives a type specification for pixel_map, which is a supertype of both GIF and PNG. To show that GIF is a subtype of pixel_map (Figure 8.2), we define the following abstraction function:

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

Using this abstraction function, the proofs that the invariant, signature, methods, and constraint rules either are straightforward to show or trivially hold. The only noteworthy aspect of pixel_map's specification is the nondeterminism specified for its *set_color* method, which is more liberal than that for both GIF images and PNG images (as we will see in Section 8.4). A call to *set_color* can always either fail

pixel_map: **type**

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

invariant *true*
constraint *true*

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

bool *set_color* (*i, j:* int; *c:* color)
modifies *p*
ensures $(result = false \wedge p_{pre} = p_{post}) \vee$
$(result = true \wedge c \in colorrange(p_{post}) \wedge c = p_{post}[i, j] \wedge$
$\forall k, l : Integer.(k \neq i \vee l \neq j) \Rightarrow p_{pre}[k, l] = p_{post}[k, l])$

end pixel_map

Fig. 8.5. A Larch Type Specification for Pixel Maps.

(making no change) or succeed (possibly adding a new color to the pixel_maps's color range). We exploit this nondeterminism later in our proofs.

8.3 Respects

8.3.1 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, K, 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 K as $A \rightarrow B$. To ensure the converter is consistent with B's specification, the specifier should show that the values of V_B to which K maps satisfy B's type invariant.

Figure 8.6 gives the definition of the *respects* relation for a converter $K : A \rightarrow B$ and type T. The first two conditions (under **Methods**) state that the original value and the converted value are indistinguishable when viewed through the methods (in particular, the observers) of type T. Let a stand for y_{pre} used in the definition (the value of the object of type A before the call to T's method m). Then the first condition requires that m's precondition holds for a's abstraction 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 postcondition holds for a's abstracted value under α iff it holds for the converted value of a abstracted under β. Thus, given

DEFINITION OF RESPECTS RELATION: Let $K : A \to B$ be a partial function mapping values of type A to values of type B. Let $A \preceq T$ and $B \preceq T$ and let $\alpha : A \to T$ and $\beta : B \to T$ be the abstraction functions for showing the corresponding subtype relations hold. Then *converter K respects T* if the following conditions hold:

- **Methods:** For each method m of T, and for all objects $x : T$, $y : A$, and $z : B$, and for all subcomputations $y_{pre}\ m_A\ y_{post}$ and $z_{pre}\ m_B\ z_{post}$, such that m_A and m_B are invocations of A's and B's corresponding methods of m, $y_{pre} \in dom(K)$, and $K(y_{pre}) = z_{pre}$:
  (i) $m.pre[\alpha(y_{pre})/x_{pre}] \Leftrightarrow m.pre[\beta(z_{pre})/x_{pre}]$ and
  (ii) $m.post[\alpha(y_{pre})/x_{pre}, \alpha(y_{post})/x_{post}]$
  $\Leftrightarrow m.post[\beta(z_{pre})/x_{pre}, \beta(z_{post})/x_{post}]$
- **Constraint:** For all integers i, j, and k, where $0 \leq i \leq j \leq k$, all histories $\rho_0 ... \rho_k$ and $\psi_0 ... \psi_k$, and for all objects $x : T$, $y : A$, and $z : B$ such that $y_{\rho_j} \in dom(K)$ and $K(y_{\rho_j}) = z_{\psi_j}$:
  $C_T[\alpha(y_{\rho_i})/x_{\rho_i}, \beta(z_{\psi_k})/x_{\rho_k}]$

Fig. 8.6. Definition of the Respects Relation.

that m is defined, then its observed state must be the same for A's values and B's values from T's viewpoint.

The last condition (labeled **Constraint**) requires that T's constraints between any two points of any history must be the same for an unconverted object of type A as for a converted object of type B. This is trivially true for two points before the conversion and for two points after the conversion; the condition more generally handles the case where one point, ρ_i, is before the conversion, and one point, ρ_k, is after the conversion, where the point of conversion is ρ_j. Thus, from T's viewpoint, the converted object's later abstracted states are consistent with the earlier abstracted states of the original object, given the history properties of T. To put it another way, now let a stand for y_{ρ_j} used in the definition. If T's history constraint holds, then an observer cannot tell that the object with the new value after conversion, $K(a)$, is any different from the object with the original value, a, before conversion.

These 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 distinguish between an object with the original value and an object with the converted value, even when taking the subsequent histories of the objects into account. Thus K *respects* T.

Claim 1 *If a and $K(a)$ abstractly map to the same value in T, i.e., $\alpha(a) = \beta(K(a))$ for all a in the domain of K, then the respects relation trivially follows.*

This special case is often useful in proofs that a converter respects a type, as we will see Section 8.4.

8.3.2 Discussion of Definition

The definition of the respects relation has a similar structure to the definition of the subtype relation. Like the subtype relation, the respects relation includes methods and constraint rules. The **Methods** rules show that the observable state of the original object and the object with the converted value are indistinguishable (from the respected type's point of view) at the time of conversion. The **Constraint** rule shows that the abstract history of the object with the converted value is consistent with the past abstracted history of the original object. Unlike the subtype relation, the respects relation does not need an invariant rule. Since both types A and B are subtypes of the respected type T, *all* objects of those types must conform to the invariants of T, by the definition of subtype, so restating the invariant rule here for particular A and B objects would be superfluous.

We considered various alternate definitions of the respects relation. One of them explicitly modeled a computation involving object $z : B$, and required it to correspond exactly to a computation involving object $y : A$, in rules like the ones above. Another did not explicitly consider histories or computations at all, but simply compared values of an object $y : A$ to values of a corresponding object $z : B$, and made sure the converted values of z matched the original values of y at every state. The first alternative was more complex than necessary; the second, too simplistic. Moreover, both failed to accommodate cases where the history of an object with a converted value diverges from that of an object with an original value, a situation we want (and need) to allow. (In the next section, we will see examples of this phenomenon. One common case occurs where one of the types in question is less constrained than the other.) The history of the original object and that of the converted object are allowed to diverge after the point of conversion, as long as both (abstracted) histories are consistent with the past (abstract) history of the original object, with respect to the respected type's constraints. As long as this consistency property holds, there should be "no surprises" from the respected type's point of view after a conversion.

8.4 Two Examples

The first example shows how our definition handles mutable methods, and hence mutable types. The second example goes one step further and shows how we handle history properties as specified in nontrivial constraint clauses.

8.4.1 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 an abstraction function that enables us to argue that PNG is a subtype of pixel_map. Then we consider converters between PNG and GIF, to argue that no total converter from PNG to GIF respects pixel_map, but that some converters from GIF to PNG do.

The type specification for PNG images is given in Figure 8.7. Note we have a nontrivial application of the renaming map, ν, where $\nu(get\_corrected\_color) = get\_color$, and $\nu(set\_corrected\_color) = set\_color$. (Only some of the methods for PNG have corresponding supertype methods; the rest are left unmapped by ν.)

We are always allowed to set a PNG image's pixels to new colors, with no limit on the total number of colors in the image; this freedom follows from the trivial type invariant and *set_corrected_color*'s specification. In contrast, we are allowed to set a GIF image's pixels to new colors only when the total number of colors does not exceed 256. The nondeterminism in the pixel_map supertype (Figure 8.5) accommodates both subtype specifications. In particular, for PNG images, the color range grows as more colors are added, so that *set_corrected_color* always successfully sets a color, if a coordinate is set within the PNG image's bounds. Moreover, although the pixel_map supertype does not have any concept of coordinate boundaries, its *set_color* method can fail for any reason, thus accommodating the behavior of PNG's *set_corrected_color* in the case that the coordinates are out of bounds.

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.

We define the following abstraction function to show that PNG is a subtype of pixel_map:

$$\alpha_P^{PM} : \mathrm{P} \rightarrow \mathrm{PM}$$

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

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

Claim 2 *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. First we show that for a given PNG

PNG: **type**

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

invariant *true*
constraint *true*

color *get_uncorrected_color* (*i, j:* int)
    requires *inframe*(p, i, j)
    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 **if** *inframe*(p, i, j)
    then *result* $= gc(p[i, j], gamma(p), \text{STDG})$
    else *result* = BLACK

bool *set_corrected_color* (*i, j:* int; *c:* color)
    modifies p
    ensures $same\_bounds\_and\_gamma(p_{pre}, p_{post}) \wedge$
    if *inframe*(p, i, j)
    then $c = gc(p_{post}[i, j], gamma(p_{post}), \text{STDG}) \wedge$
        $c \in colorrange(p_{post}) \wedge$
        $\forall k, l : Integer.(k \neq i \vee l \neq j) \Rightarrow p_{pre}[k, l] = p_{post}[k, l])) \wedge$
        $result = true$
    else $p_{pre} = p_{post} \wedge result = false$

end PNG

Fig. 8.7. A Larch Type Specification for PNG Images.

value, p, where $|\ colorrange(p)\ | = n$ *and* $n > 256$, *its abstracted pixel_map value,* $\alpha_P^{PM}(P)$, *also has at least 256 colors. From the abstraction function,* $\alpha_P^{PM}(p)[i, j] = gc(p[i, j], gamma(p), STDG)$, *we know that 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 have to be the same, that is,*

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

By the "transitivity" and "reflexivity" properties of gamma correction functions (see the Appendix),
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 colors in $\alpha_P^{PM}(p)$.

Next we show that the conversion of p to a valid GIF image value $K(p)$ would cause the GIF value to be observably different from the PNG value, when viewed through pixel_map*'s interface. $K(p)$ can have a maximum of 256 colors by the type invariant of GIF image. Furthermore, the abstraction mapping of $K(p)$ to a pixel_map value, $\alpha_G^{PM}(K(p))$, cannot add any colors to $colorrange(K(p))$ (except for BLACK), since we see from the definition of α_G^{PM}, and hence by the definition of overlay, that every $c \in colorrange(\alpha_G^{PM}(K(p)))$ is either BLACK or one of the colors used in one of the frames of $K(p)$. Therefore, there exists some color c such that $c \in colorrange(\alpha_P^{PM}(p))$ and $c \notin colorrange(\alpha_G^{PM}(K(p)))$. It follows that there exists some i and j such that the result of calling* pixel_map*'s observer,* get_color *with arguments i and j will differ between a call on the original PNG image from a call on the converted GIF image, that is, $\alpha_P^{PM}(p)[i,j] \neq \alpha_G^{PM}(K(p))[i,j]$ Therefore, the converter cannot respect pixel_map.* □

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

Claim 3 *There exist converters from GIF to PNG that respect pixel_map.*

Proof: By existence. Here is a converter from GIF to PNG:

$K : G \to P$
$K(g) = p$ *where*
$xmin(p) = min(\{xmin(g[i]) \mid 0 \leq i < len(g)\}) \wedge$
$xmax(p) = max(\{xmax(g[i]) \mid 0 \leq i < len(g)\}) \wedge$
$ymin(p) = min(\{ymin(g[i]) \mid 0 \leq i < len(g)\}) \wedge$
$ymax(p) = max(\{ymax(g[i]) \mid 0 \leq i < len(g)\}) \wedge$
$gamma(p) = STDG \wedge$
$\forall i, j : \text{Integer} \;.\; p[i,j] = overlay(g, i, j)$

Composing our converter with our abstraction function from PNG to pixel_map, for an original gif value, g, we get an abstracted converted pixel_map value, pm, such that

$$\forall i, j : \text{Integer} \;.\; pm[i,j] = \begin{cases} gc(overlay(g,i,j), STDG, STDG) \\ \quad if\, min(\{xmin(g[k]) \mid 0 \le k < len(g)\}) \le i \wedge \\ \quad\quad i \le max(\{xmax(g[k]) \mid 0 \le k < len(g)\}) \wedge \\ \quad\quad min(\{ymin(g[k]) \mid 0 \le k < len(g)\}) \le j \wedge \\ \quad\quad j \le max(\{ymax(g[k]) \mid 0 \le k < len(g)\}) \\ BLACK, otherwise \end{cases}$$

By the "reflexivity" property of gamma correction functions, we know that

$$gc(c, g, g) = c.$$

Furthermore, we know that from the definition of overlay that when i and j are beyond the bounds of any frames in a frameset, $overlay(g,i,j)$ is $BLACK$. In the definition above, whenever i or j are outside the respective minima or maxima, then (i,j) is outside any frame in the GIF. Therefore the definition above simplifies to

$\forall i, j : \text{Integer} \;.\; pm[i,j] = overlay(g,i,j)$

which is exactly the same abstraction function as is used to map the original GIF to a pixel map. The abstracted values are identical, so by Claim 1 (made at the end of Section 8.3.1), the conversion respects pixel_map. □

Our assertion above may seem to go against our intuition, when we consider the further histories of the original GIF and the converted PNG. After all, the same sequence of mutations called at the pixel_map level can cause the histories of the original object and the converted object to diverge. Indeed, our type specifications for GIF and PNG mandate that the histories sometimes *must* diverge. Consider, for instance, an object o which at some state ρ has exactly 256 colors. Attempting to call *set_color* (within appropriate x and y bounds) with a 257th color *must* fail for the original GIF object, since it has a maximum of 256 colors. However, it *must* succeed for the converted PNG object, since *set_color* as defined on PNG images cannot fail if it is called within the x and y bounds of the image. From this point on, then, the histories of the original GIF image and the converted PNG image diverge.

For our conversion to respect pixel_map, however, it suffices that the GIF image and the converted PNG image, at the time of conversion, have identical *possible* futures from pixel_map's perspective. That is, we should not be able to tell, given only the mutations and requirements of pixel_map, that the object with the original value and the object with the converted value were different at the point of conversion. As long as this is true, programs expecting to operate on a pixel_map object will not encounter surprises if the object they operate on had been converted from a GIF to a PNG image.

Our pixel_map type has only one mutator, *set_color*. All we know about *set_color* is that any attempt to add a new color to a given pixel_map might succeed or might fail. Either outcome is possible from pixel_map's perspective, no matter how many

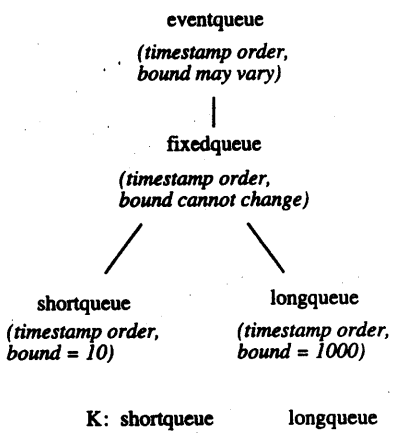

Fig. 8.8. A Queue Hierarchy.

colors are in a pixel map at a given time. So, from pixel_map's perspective, any sequence of pixel_map method calls on both an original GIF object and a converted PNG object will have the same possible future observed behaviors. Since the possible future histories of the PNG and GIF objects look the same from pixel_map's point of view, there will be no surprises when converting from GIF to PNG, if one assumes only the behavior specified in pixel_map.

8.4.2 Event Queues

The types in the previous example had invariants but no history constraints. In this section's example of an event queue type family (Figure 8.8), we look at a conversion between constrained types, and show which common supertypes the conversion respects, and which it does not.

At the root of the type hierarchy, we have an eventqueue type that models buffered event queues (Figure 8.9). We represent a value of an eventqueue object, q, as a pair, [*items, bound*], of a set of the buffered items and a bound. Events in the queue must be inserted in increasing timestamp order. The size of the queue buffer is bounded, but the bound is not directly readable or writable by the eventqueue type. New events (if they have appropriate timestamps) can be inserted into the queue unless the number of items already in the queue is equal to (or greater than) the bound. The eventqueue type also has the overall constraint that the event at the head of the queue at a state ρ_i has to have a timestamp less than or equal to the event at the head of the queue at any later state ρ_k. The constraint, however, does not require that its bound be fixed, so the bound can vary over time. (The specification subtly allows this possible mutation since the *insert* and *remove* methods each has

eventqueue = **type**

uses EventQueue (eventqueue **for** Q)
for all q: eventqueue

invariant $len(q_\rho.items) \leq q_\rho.bound$
constraint $timestamp(head(q_{\rho_i}.items)) \leq timestamp(head(q_{\rho_k}.items))$

bool *insert* (*e:* event)
requires $timestamp(last(q_{pre}.items)) < timestamp(e)$
modifies q
ensures $len(q_{post}.items) \leq q_{post}.bound \wedge$
if $len(q_{pre}.items) < q_{post}.bound$
then $q_{post}.items = add(e, q_{pre}.items) \wedge result = true$
else $q_{post}.items = q_{pre}.items \wedge result = false$

event *remove* ( )
requires $q_{pre}.items \neq empty$
modifies q
ensures $q_{post}.items = tail(q_{pre}.items) \wedge result = head(q_{pre}.items) \wedge$
$len(q_{post}.items) \leq q_{post}.bound$

int *size* ( )
ensures $result = len(q_{pre}.items)$

end eventqueue

Fig. 8.9. A Type Specification for Event Queues.

a **modifies** clause; though no mention of changing the queue's bound is made in either of their postconditions, the presence of the **modifies** clause gives permission to implementors to change that part of the queue's value and simultaneously warns callers that they cannot rely on that part of the queue's value to remain the same. Even more subtly, the condition in the **if** ... **then** ... **else** clause for *insert* compares the length of the buffer of the queue's pre-state with the bound of the queue's *post*-state to account for the possibility of the bound changing as a side effect of calling *insert*.)

Let fixedqueue (Figure 8.10) be a subtype of eventqueue which adds the further constraint that the buffer bound cannot change. Again, there are no methods to read the bound directly. Two subtypes of fixedqueue, shortqueue and longqueue, further specify that the bound is fixed to be 10 and 1000 items, respectively (Figures 8.11 and 8.12). The **subtype** clause in a type specification includes an abstraction function, α, that relates subtype values to supertype values. Implicitly the clause requires that the subtype provides all methods of its supertype; any method not

fixedqueue = **type**

uses EventQueue (fixedqueue **for** Q)
for all q: fixedqueue

invariant $len(q_\rho.items) \leq q_\rho.bound$
constraint $timestamp(head(q_{\rho_i}.items)) \leq timestamp(head(q_{\rho_k}.items)) \wedge$
$q_{\rho_i}.bound = q_{\rho_k}.bound$

bool *insert* (*e:* event)
requires $timestamp(last(q_{pre}.items)) < timestamp(e)$
modifies q
ensures $q_{pre}.bound = q_{post}.bound \wedge$
if $len(q_{pre}.items) < q_{post}.bound$
then $q_{post}.items = add(e, q_{pre}.items) \wedge result = true$
else $q_{post}.items = q_{pre}.items \wedge result = false$

event *remove* ( )
requires $q_{pre}.items \neq empty$
modifies q
ensures $q_{post}.items = tail(q_{pre}.items) \wedge result = head(q_{pre}.items) \wedge$
$q_{post}.bound = q_{pre}.bound$

subtype of eventqueue
$\forall q : Q \;.\; \alpha(q) = q$

end fixedqueue

Fig. 8.10. A Type Specification for Fixed Queues.

renamed or redefined is "inherited" from its supertype as is. For example, shortqueue's *insert* does not redefine fixedqueue's *insert*, but fixedqueue's does redefine eventqueue's.

Claim 4 *There is no conversion from shortqueue to longqueue, either partial or total, that respects fixedqueue.*
Proof: Consider a shortqueue object s in the domain of the conversion. By the definition of shortqueue, its bound must be 10. Suppose a conversion is made of s_{ρ_j}, yielding a longqueue value l_{ψ_j}. By the definition of the longqueue type, the bound of the converted object must be 1000. Now consider s_{ρ_i} prior to the conversion and the value l_{ψ_k} after the conversion. The constraint of fixedqueue is violated, since $s_{\rho_i}.bound \neq l_{\psi_k}.bound$, and fixedqueue's constraint prohibits the bound from changing between states. The **Constraint** *condition of the definition of respects does not hold. Hence, the conversion cannot respect fixedqueue.* □

The histories of the converted object show the failure of the conversion to respect

shortqueue = **type**

uses EventQueue (shortqueue **for** Q)
for all q: shortqueue

invariant $len(q_\rho.items) \leq q_\rho.bound \wedge q_\rho.bound = 10$
constraint $timestamp(head(q_{\rho_i}.items)) \leq timestamp(head(q_{\rho_k}.items))$

subtype of fixedqueue
$\forall q : Q \;.\; \alpha(q) = q$

end shortqueue

Fig. 8.11. A Type Specification for Short Queues.

longqueue = **type**

uses EventQueue (longqueue **for** Q)
for all q: longqueue

invariant $len(q_\rho.items) \leq q_\rho.bound \wedge q_\rho.bound = 1000$
constraint $timestamp(head(q_{\rho_i}.items)) \leq timestamp(head(q_{\rho_k}.items))$

subtype of fixedqueue
$\forall q : Q \;.\; \alpha(q) = q$

end longqueue

Fig. 8.12. A Type Specification for Long Queues.

fixedqueue. While it is possible to define a simple conversion from shortqueue to longqueue that contains exactly the same items, the conversion of its bound (from 10 to 1000) changes possible future behaviors of the longqueue object in ways not consistent with the fixedqueue constraints. Suppose, for instance, that we have a program that fills up a queue buffer to determine its size, and uses this information to allocate a fixed-size buffer of its own to store items pulled off the queue. At some later point, after more items have been added to the queue, the program empties the queue items into its own fixed-size buffer. If the queue bound has been increased by a conversion, the program may overflow its previously-allocated buffer, causing a crash or other errors.

To illustrate the incongruity above, we must track the behavior of the original

object and the converted object over time, through the point of conversion. It is not enough simply to look at each state and to compare the original object's value and the converted object's value in that same state. In the queue example, the longqueue type is less constrained than the shortqueue type, and so some possible longqueue values are outside the range of any converter on shortqueue values. Once a conversion takes place, we need to reason about the object in terms of its longqueue values; moreover, it would be ill-defined in subsequent states to compare its longqueue value to any shortqueue value. However, we can allow such nonsurjective converters as long as the converted object's value does not violate constraints of the respected type.

For instance, it is possible to convert from shortqueue to longqueue in a way that respects the more general eventqueue type.

Claim 5 *There is a total conversion from shortqueue to longqueue that respects eventqueue.*

Proof: By existence. Here is such a converter:

$K : Q \to Q$
$K(q) = [q.items, 1000]$

To see whether this conversion respects eventqueue, we first check the method rules. The pre- and postconditions of the methods remove *and* size*, and the precondition of the method* insert*, depend only on the items portion of the queue value, which is the same for both a shortqueue object s and a longqueue object l at the time of conversion. The* insert *method's postcondition depends in part on the post-state of bound, but since eventqueue allows bound to change on insert, eventqueue's specification permits either s or l's bound to change, and hence allows the operation to insert an item or not for either object. (While s and l will in fact behave differently based on the more constrained specifications of shortqueue and longqueue, the eventqueue specification itself cannot be used to tell that anything changed in the conversion.)*

The only constraint of eventqueue is that the timestamp at the head of the queue not decrease over time. Let s_{ρ_i} *be any value of s before the conversion,* s_{ρ_j} *be the value of s at the time of conversion,* l_{ψ_j} *be the value of l at the time of conversion, and* l_{ψ_k} *be any value of l after the conversion. Since shortqueue and longqueue are both subtypes of eventqueue, which includes the history constraint, we know that*

1. $timestamp(head(s_{\rho_i}.items)) \leq timestamp(head(s_{\rho_j}.items))$ *and*
2. $timestamp(head(l_{\psi_j}.items)) \leq timestamp(head(l_{\psi_k}.items))$

Furthermore, by the definition of our conversion,

3. $s_{\rho_j}.items = l_{\psi_j}.items$

so therefore

4. $timestamp(head(s_{\rho_j}.items)) = timestamp(head(l_{\psi_j}.items))$

By transitivity, then,

5. $timestamp(head(s_{\rho_i}.items)) \leq timestamp(head(l_{\psi_k}.items))$

Hence, the constraint rule is satisfied. Therefore, the conversion, K, as a whole respects eventqueue. □

Again, this result matches our expectations. The buffer-overflowing program mentioned earlier got into trouble only because the programmer assumed that the eventqueue's buffer bound would not change, and therefore allocated a fixed buffer for receiving events from the queue. Programmers that do not assume that the eventqueue's buffer bound is fixed should allow for arbitrarily many events to be emptied from the queue, and thus avoid the error of the previous program.

8.5 Related Work

There are notions of "respectful" conversions that are stronger or weaker than the one we present in this chapter; they may be more appropriate in certain situations.

In earlier work [WO99a], we gave a simpler model of respectful conversion for immutable types. Since the objects are immutable, there are no mutators or history properties to consider; hence, the predicates under the **Methods** condition are simpler and the **Constraint** condition is entirely unnecessary. We showed in that paper how the simpler model is useful for applications that retrieve and analyze data. When applied to mutable objects, however, the failure to consider history properties may produce behavior in the converted object that is inconsistent with the past behavior of the original object, as we saw in our queue example.

A longer version [WO99b] of this chapter contains informal descriptions of other real-world examples, including those used in the TOM context and those addressing the Y2K problem; it also contains a formal proof of the "no surprises" claim for pixel_map made at the very end of Section 8.4.1.

Applications that pass converted data back and forth between heterogeneous programs may require stronger guarantees on conversions. For example, the Mockingbird system [ACC97], defines a notion of *interconvertibility* where data conversions are fully invertible. With this policy, data converted to another format can always be converted back without loss of information. This concept corresponds in our model to a conversion between two representations of some type T, where the conversion respects type T. While interconvertibility makes it easy to exchange transformed data without risk of losing information, it is often too strong a constraint on conversions. Our model of respectful type conversions is more flexible. In respectful conversions, some information can be lost or changed in the conversion, provided that the information and behavior of the respected type is preserved.

8.6 Summary

In this chapter, we extended our earlier definition of a novel notion of *respectful type converters* to capture what behavior a conversion function preserves when transforming objects of one type to another; our extension deals with mutable types. We

built on Liskov and Wing's notion of behavioral subtyping. Our notion of respectful type converters could probably be adapted for any other notion of subtyping (behavioral or not, dealing with mutable types or not).

Analyzing the types that a conversion respects allows developers to determine where programs will continue to behave normally after data is converted, and where they may behave unexpectedly or erroneously. Intuitively, if a conversion respects a type T, then after an object of type A is converted to an object of type B in a conversion that respects T, programs that operate on the objects using the interface and expectations of T will encounter no surprises. Programs that use more detailed interfaces or that rely on behavioral assumptions specified by A or B but not by T, however, may encounter problems. Reviewing the assumptions programs make about data and seeing what types conversions respect allow us to detect possible conflicts introduced by converted data, and to adjust programs appropriately.

Bibliography

[ACC97] Auerbach, J. and Chu-Carroll, M. C. The Mockingbird System: A Compiler-based Approach to Maximally Interoperable Distributed Programming. Technical Report RC 20718, IBM, Yorktown Heights, NY, 1997.

[GG89] Garland, S. and Guttag, J. An Overview of LP, the Larch Prover. In *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 137–151, Chapel Hill, NC, April 1989. Lecture Notes in Computer Science 355.

[HGJ+93] Horning, J., Guttag, J. w. S. G., Jones, K., Modet, A., and Wing, J. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, 1993.

[Jon98] Jones, C. Bad Days for Software. *IEEE Spectrum*, 35(9):47–52, September 1998.

[Lis87] Liskov, B. Data Abstraction and Hierarchy. In *OOPSLA'87: Addendum to the Proceedings*, 1987.

[LW94] Liskov, B. and Wing, J. M. A Behavioral Notion of Subtyping. *ACM TOPLAS*, 16(6):1811–1841, November 1994.

[Ock98] Ockerbloom, J. Mediating Among Diverse Data Formats. Technical Report CMU-CS-98-102, Carnegie Mellon Computer Science Department, Pittsburgh, PA, January 1998. Ph.D. Thesis.

[Win97] Wing, J. Subtyping for Distributed Object Stores. In *Proceedings of the Second IFIP International Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, pages 305–318, July 1997.

[WO99a] Wing, J. M. and Ockerbloom, J. Respectful Type Converters. *IEEE Transactions on Software Engineering*, 1999. to appear.

[WO99b] Wing, J. M. and Ockerbloom, J. Respectful Type Converters for Mutable Types. Technical Report CMU-CS-99-142, Carnegie Mellon Computer Science Department, June 1999.

[Zar96] Zaremski, A. M. Signature and Specification Matching. Technical Report CS-CMU-96-103, CMU Computer Science Department, January 1996. Ph.D. Thesis.

Appendix: Larch Traits and Type Specifications

This appendix contains the following Larch specifications: Color trait for color literals, ColorSet trait for sets of colors, Frame trait, FrameSeq trait, GIFImage trait, Gammas trait, PNGImage trait, PixelMap trait, Event trait, event type, and EventQueue trait. Appendix A of the Larch Book [HGJ+93] contains traits for Boolean, Integer, FloatingPoint, Set, Deque, Array2, TotalOrder, and Queue, all of which we use below.

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 → Integer
    colorrange : F → CS
    inframe : F, Integer, Integer → Boolean
  asserts for all i, j : Integer, f : F
    $xmin(f) \leq xmax(f)$
    $ymin(f) \leq ymax(f)$
    $inframe(f,i,j) = (xmin(f) \leq i \leq xmax(f)) \wedge (ymin(f) \leq j \leq ymax(f))$
    $inframe(f,i,j) \Rightarrow f[i,j] \in colorrange(f)$
end Frame

FrameSeq(F, FS): **trait**
  includes Deque (Frame, FS)
  introduces
    overlay: FS, Integer, Integer → Color
    changepixel: FS, Integer, Integer, Color → FS
    colorrange: FS → CS
    _[_]: FS, Integer → F
  asserts for all *i, j, k, l:* Integer, *c:* Color, *f:* F, *fs:* FS
    overlay(fs, i, j) = **if** *len(fs)* = 0 **then** BLACK **else**
        if $inframe(last(fs), i, j)$
        then $last(fs)[i,j]$
        else $overlay(init(fs), i, j)$
    $colorrange(empty) = \{\}$
    $colorrange(fs \vdash f) = colorrange(fs) \cup colorrange(f)$
    $(fs \vdash f)[i]$ = **if** $i = len(fs \vdash f)$ **then** f **else** $fs[i]$
    $overlay(changepixel(fs, i, j, c), k, l)$ = **if** $i = k \wedge j = l$ **then** c **else** $overlay(fs, k, l)$
  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
same_bounds_and_gamma: P, P $\rightarrow$ Boolean
asserts for all p, q: P
$same\_bounds\_and\_gamma(p, q) = (gamma(p) = gamma(q) \wedge$
$xmin(p) = xmin(q) \wedge xmax(p) = xmax(q) \wedge ymin(p) = ymin(q) \wedge ymax(p) = ymax(q))$
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

Event: **trait**
includes TotalOrder (Time **for** T)
introduces
timestamp: Ev $\rightarrow$ Time

event: **type**
uses Event (event **for** Ev)
end event

EventQueue: **trait**
includes Queue (Ev **for** E), Event
Q **tuple of** items: C, bound: Integer