# Unshareable Dynamic Objects and How to Resolve a Conflict Between Encapsulation and Pointers

Naftaly H. Minsky*
Computer Science Department
Rutgers University
New-Brunswick, NJ 08903
Tel: (908) 445-2085; e-mail: minsky@cs.rutgers.edu

January 1995

**Abstract**

This paper introduces the concept of an *unshareable dynamic object*, i.e., an object that is created dynamically on the heap, and is guaranteed to have a *unique* and *movable* pointer leading to it. The use of such unshareable objects, whenever sharing is not required, is expected to fortify the concept of encapsulation, to make systems simpler and easier to reason about, and to make storage management safer and more efficient. We argue that unshareable objects can be implemented by means of few minor and virtually costless modifications in conventional OO languages, and we demonstrate this in detail for the Eiffel language.

Category: research
Topic areas: language design, architecture

*Work supported by NSF grant No. CCR-9308773.

# 1 Introduction

*Dynamic objects*, i.e., objects allocated on the heap and addressed by means of pointers, are widely considered a *necessary evil* in imperative programming. Necessary, because they provide some very important capabilities, and evil, because they make reasoning about systems much more difficult and storage management more hazardous and costly. The useful capabilities provided by dynamic objects are largely as follows:

1. *Dynamic creation* and indefinite scope and life time.
2. *Efficient transfer* from one place in a system to another, simply by transferring pointers.
3. *Shareability*, via multiple pointers to a single object.

Of these, it is the shareability of dynamic objects which is problematic. It allows for many aliases to exist for a given dynamic object, anywhere in the system, making it hard to reason about this object. It is, in particular, risky to deallocate an object, due to the specter that its aliases would become *dangling references* — one of the main reasons for languages to employ garbage collection. Moreover, under conventional programming languages it is impossible to utilize the harmless capabilities of dynamic objects without incurring the pitfalls of their shareability, whether or not any sharing is actually desired. This is particularly serious for object-oriented programming with its heavy reliance on dynamic objects.

To alleviate this drawback, we introduce in this paper a concept of an *unshareable dynamic object*. This is an object which is created dynamically on the heap but which can have only one (movable) pointer leading to it. The use of such unshareable objects instead of the conventional heap objects, whenever sharing is not required, should make systems easier to reason about, and storage management safer and more efficient, while retaining all the harmless capabilities of conventional dynamic objects.

The implementation of unshareable objects to be proposed here rests on a departure from the almost universal use of *copying* (the copying of pointers, in the case of dynamic objects) as the means for transferring information from one place in the system to another. Generally speaking, we propose that objects designated as unshareable be transferred *by move* rather than *by copy*.

For the sake of specificity we couch our discussion in terms of the object-oriented language Eiffel [6], describing in detail how unshareable objects can

be implemented by means of few minor, and virtually costless, modifications of this particular language. But we believe that the essence of our conclusions is valid for many other object-oriented languages, and, in a broader sense, is applicable to imperative languages in general.

This paper is organized as follows: In Section 2 we demonstrate the pitfalls of conventional dynamic objects by showing that their use seriously compromises the principle of *encapsulation*. In Section 3 we define our concept of unshareable dynamic objects; in Section 4 we show how the Eiffel language can be modified to support this concept; in Section 5 we discuss some related work, including recent work on "linear objects" in the context of functional programming; and we conclude in Section 6 with a brief discussion of the applications of unshareable objects, and their potential implications to storage management.

# 2 The Conflict Between Encapsulation and Dynamic Objects

To demonstrate some of the pitfalls of dynamic objects, we show here their harmful effect on the principle of encapsulation — the bedrock of object-oriented programming. This effect is, briefly, as follows: encapsulation calls for the component parts of an object to be *hidden*, and thus protected from any influence from the outside. But when a component of an object is defined not by containment but by a pointer, it is, in a sense, exposed to outside influence, undermining the raison d'etre of encapsulation. We will first define the sense in which dynamic objects are hard to hide, and then discuss the effect of this phenomenon on encapsulation.

## 2.1 Dynamic Object are Hard to Hide

The concept of "hiding" in software is somewhat slippery, and may have several reasonable definitions that reflect different aspects of it. Here is one such definition, whose significance will become evident in the following section:

**Definition 1 (the concept of hiding)** *A component* `c` *of object* `x` *is considered* hidden *in* `x` *only if it is not accessible (from anywhere) while* `x` *does not have control. (*`x` *is said to* have control *between the invocation of one of its methods, and the return from this method.)*

If a component `c` of an object `x` is physically *contained* in it, as illustrated in part (a) of Figure 1, then this condition can be readily established by the

scope rules of the language, as it is in the case of Eiffel, in particular.[1] But if c is a dynamic object, addressed via a variable p_c contained in x then, the scope rules are not sufficient to hide it. Indeed, even if variable p_c is not visible from the outside, the object c itself is quite exposed to any object that may have a pointer to it, as illustrated by part (b) of Figure 1. Any such object may operate on c even when x does not have control, in direct contradiction to the above definition of hiding.

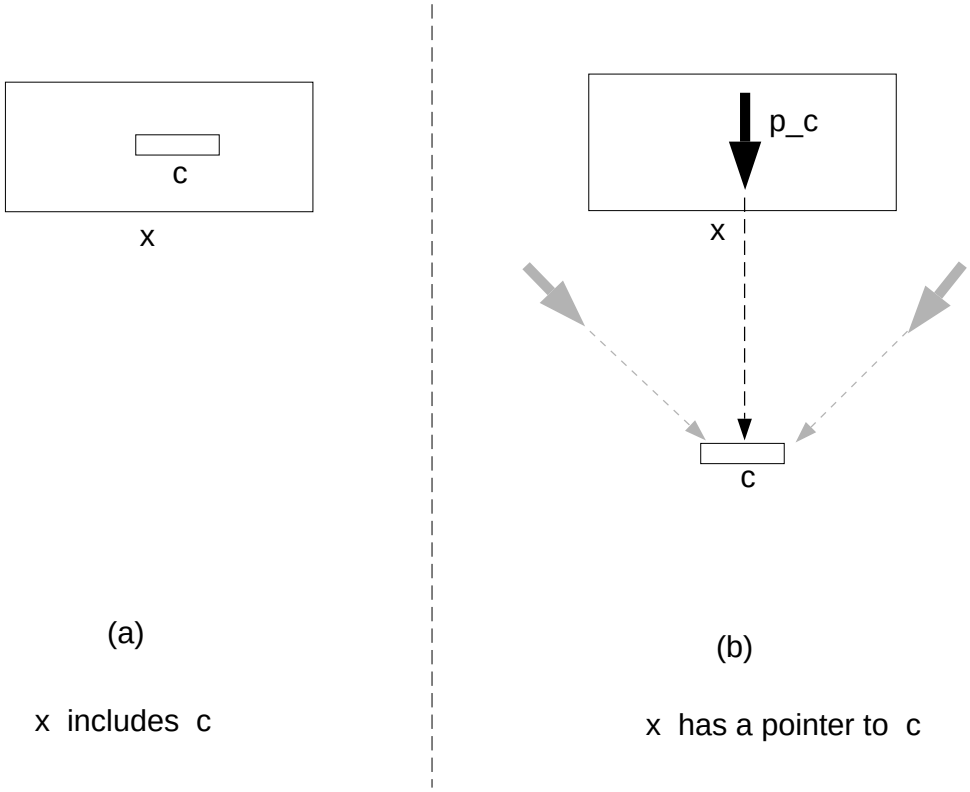


Figure 1: The effect of shareability on encapsulation

Moreover, it is virtually impossible for an object to prevent or control the distribution of pointers to its dynamic component parts. The reason is that the transfer of data from one place in a system to another is carried

[1] Actually, even the hiding of such components is rarely, if ever, *completely* ensured, because of the *unsafe* features [2] that most languages have, such as the ability to use naked C-code in Eiffel and in C++. We ignore the effect of such unsafe features in this paper.

out, almost universally, *by copy* — the copy of pointers when dealing with dynamic objects. This is the case for the *assignment statement*

```
u := v;
```

which copies the pointer in `v` into `u`, leaving `v` intact — thus creating a duplicate of this pointer. (Note that in Eiffel assignments have "reference semantics," when applied to reference variables, i.e., to variables that hold references to dynamic objects.) This is also the case with the *implicit assignment* that takes place during parameter passing.

Therefore, coming back to our example, if `x` obtained its component `c` (that is, the pointer to it) from some other object, then `x` cannot tell if there are any pointers to `c` left elsewhere in the system. Even if `x` itself is the original creator of `c`, there is very little it can do to prevent the leakage of pointers for `c` into other objects in the system. This, because almost anything that `x` does with `p_c`, would provide other objects with the opportunity to acquire a duplicate pointer to `c`. For example, a procedure call

```
y.f(...,p_c,...),
```

carried out by `x`, allows procedure `f` to save a pointer for `c` permanently in some attribute of object `y`.

Note that although our definition of hiding is strictly stronger than hiding by scope rules, it allows for a component `c` of an object `x` to be accessed by other objects, *as long as control is in* `x`. For example, `x` may invoke operation `y.p(c)`, thus having procedure `p` of object `y` operate on `c`. (This is one sense in which the concept of hiding is slippery.)

## 2.2 The Difficulty in Establishing Invariants

We distinguish between two kinds of benefits that encapsulation is reputed to provide. First, it is supposed to enable us to endow an object (or, a class of objects) with what is often called *invariants* (or *class invariants*). These are properties that *"holds whenever control is not in the object"* (Sethi ([8]). Second, encapsulation is supposed to provide objects with *implementation transparency*; i.e., the ability to change the internal representation of the state of an object, while maintaining its invariants, without having to change anything in the rest of the system. It is the former of these benefits of encapsulation which is compromised when the components of an object are dynamic, as is illustrated by the following example.

Let an object x have a collection of components $c_1, ..., c_k$, and let each such component have a `weight` attribute, which can be modified by method `set_weight` defined for all these objects. Suppose that x is required to have the following *invariant* property:

> *The total weight of all the components of* x *does not exceed a given limit* w.

Now, if $c_1, ..., c_k$ are dynamic components, it is impossible to ensure even this simple property as an invariant of x, because these components are not hidden, by our definition of this term. In other words, any of the components $c_i$ of x may be shared by (accessible to) some other object y, which may raise the weight of this component when x is not in control.

This is a serious problem because invariants properties are the basis for meaningful modularization and for abstract data types. Yet, although this problem with encapsulation is not unknown (see [5] page 159, in particular) it is rarely discussed in literature, and has not been satisfactorily resolved so far. The lack of attention to this difficulty may be due to the fact that the second reputed benefit of encapsulation, i.e., with *implementation transparency*, is not effected by dynamic objects. In our example above, in particular, the internal organization of object x, e.g., the data structure that maintain the collection of pointers to components $c_1, ..., c_k$, is still hidden from the outside, and can be changed without changing the rest of the system.

# 3 The Concept of Unshareable Dynamic Objects

Since the very shareability of dynamic objects is so harmful it should be useful to be able to avoid it whenever sharing is not needed, without loosing any of the other advantages of such objects. This leads us to the following concept:

**Definition 2 (unshareable objects)** *A dynamic objects is called* unshareable *if it is guaranteed to have only one, movable, pointer leading to it.*

Broadly speaking, objects can be made unshareable by making sure that pointers to them are never duplicated when transferred from one variable to another.[2] This would involve mainly changes in the semantics of assignment

[2] In a language like C++ one would also have to prohibit the creation of new pointers for unshareable objects; namely the application of the & operator to such objects.

statements and of parameter passing, the main transfer mechanisms in programming languages. Assignment can be performed simply by *moving* the pointer in question from the right hand side of the assignment statement to the variable at its left hand side. But such a simple move-semantics is inappropriate for parameter passing, as illustrated by the following example.

Consider an object `x` that has an unshareable component `u`, and let `t` be an object that represents a user-terminal. If parameter passing is done by move, then the method invocation

```
t.display(u),
```

would *move* the pointer in `u` into the corresponding formal argument of procedure `display`, leaving a null pointer in `u`. In a sense, `u` would be *consumed* by procedure `display`, which is clearly not an intended side effect of this procedure call. Moreover, `u` will be consumed by any operation `u.m` on it with one of its own methods, because, as will be argued later, such an operation actually supplies `u` as a parameter to procedure `m`. Such consumption of unshareable objects, almost anytime they are used, is usually undesirable and would make the very concept of unshareable objects quite untenable for object-oriented programming.

To prevent procedures from consuming their unshareable parameters we propose that such parameters be passed *by lending*. By this we mean that a call

```
y.r(...,u,...),
```

where `u` points to an unshareable object, *moves* the pointer in `u` into the corresponding formal parameter for the duration of the lifetime of procedure `r`, *to be returned* to `u` upon the completion of this procedure.

Some procedures, however, may be intentionally designed to consume their unshareable parameters. For example, if our object `x` performs the operation

```
s.push(u),
```

where `s` is a stack, then `u` *must* be consumed to be stored in the stack, if it points to an unshareable object. To provide for such consumption by a procedure, we allow for an unshareable parameter to be declared as *consumable*, resulting in a weaker version of the "lending" of such parameters.

To summarize, then, we propose that pointers to unshareable objects be assigned *by move*, and passed as parameters *by lending* (either strong lending or weak lending). We show how this can be accomplished in Eiffel, next.

# 4 Unshareable Objects Under Eiffel

In this section we show how the object-oriented language Eiffel can be made to support unshareable objects. Technically, we describe a variant of Eiffel, obtained by a small set of minor modifications of the semantics of this language. We refer to this variant as Eiffel*, but what we really advocate here is that the Eiffel language itself be changed to meet these rules, and that analogous changes be made in other object-oriented languages.

The modifications in Eiffel advocated here will be referred to as the *rules* of Eiffel*. The compile-time cost and the run-time overhead required to establish these rules turns out to be quite negligible. Moreover, these rules impose no constraint on anything not involving unshareable objects. Therefore an Eiffel* program that does not use unshareable objects is equivalent to a standard Eiffel program.

A disclaimer is in order here: The assurance provided by by Eiffel* that objects designated (in a manner to be discussed below) as unshareable actually satisfy Definition 2, is not absolute. It is at about the same level of the certainty that Eiffel provides for type correctness and for its scope rules. All such assurances are not absolute because Eiffel, like practically all other languages, has some *unsafe features* which if used carelessly may violate the semantics of the language itself, as already mentioned in Footnote 1.

The rest of this section is organized as follows: We start by introducing a concept *unshareable variables* which are used to address unshareable objects. This is followed with rules that define the treatment of such variables by the assignment statements, by parameter passing, and by some other constructs of Eiffel. In Section 4.5 we show how unshareable objects can be recycled efficiently, and we conclude in Section 4.6 with a brief discussion of the use of such objects.

## 4.1 Unshareable Variables

Unshareable objects are addressed in Eiffel* by means of variables declared as `unshareable`. Such variables may be used to represent *attributes* of an object, *parameters* and *local variables* of a procedure, or the implicitly defined `result` variable of a function. Unshareable variables are referred to as *u-variables*, and are often named by symbols such as `u`, `u1`, `v` and `v1`. As we shall see, u-variables are guaranteed to contain *unique* pointers (or the value `void`), which they ought to if they are to point to an unshareable object. The pointers contained in u-variables are sometimes called *u-pointers*.

Generally speaking, variables of any class can be declared as unshareable,

subject only to a restriction imposed by Rule 6 (to be introduced later). For convenience, we allow for a class to be declared as unshareable, which would mean that a variable declared to be of this class is unshareable by default.

The first rule of Eiffel* prevents pointers from being transferred from conventional variables into u-variables, or the other way around, as is stated below:

**Rule 1** *Mixed assignments and mixed parameter passing, involving a conventional variable and a u-variable, are not permitted.*

This would mean that every object created via a u-variable (i.e., by means of an instantiation statement such as `!!u`) is an unshareable object. This is because due to the above rule, a pointer to this object can be transferred only to u-variables, and because, as we shall see, the pointers contained in u-variables cannot be duplicated. The rest of the rules of Eiffel* deal with the treatment of u-variables by various constructs of this variant of Eiffel.

### 4.2 Assignment of Unshareable Variables is *by Move*

The assignment statement in Eiffel has *reference semantics*, when applied to reference variables; i.e., it is a pointer which is copied by an assignment, not the object being pointed to. The following rule, which applies to the assignment of u-pointers, causes such pointers to be moved by the assignment, instead of being copied.

**Rule 2** *An assignments statement* `u2 := u1` *is carried out as follows: first, the value of* `u2` *is copied into* `u1`*; and, second, if* `u1` *is a variable then it is nullified; i.e., the value* `void` *(the null pointer of Eiffel) is stored in it.*

In other words, if the right-hand side of an assignment is a u-variable then its pointer *moves* to the left-hand u-variable, leaving void in its wake. If the right hand side of an assignment is a function (which, must, then, return a u-pointer) then the variable that contains the value of this function disappears automatically, along with its activation record, and is of no concern to us here. Note that the assignment statement is also subject to the optional Rule 11, which deals with the deallocation of unusable objects.

### 4.3 The Passing of Unshareable Parameters is *by Lending*

Consider a procedure call

```
x.r(...,u,...),
```

where u is an unshareable variable. Let p be the pointer residing in u, and let v be the formal parameter of r that corresponds to u. Informally, we say that u is passed *by lending*, if pointer p of u *is moved* into v for the duration of the lifetime of procedure r, to be returned to u upon the completion of r. This kind of parameter passing is defined by the following two rules, the first of which deals with the parameter passing mechanism itself, and the second deals with the treatment of formal unshareable parameters by their procedures.

**Rule 3** *The passing of an unshareable parameter* u *to the corresponding formal parameter* v *is carried out as follows: (a) the pointer in* u *is* moved *into* v *before the procedure gets control; and (b) if* u *is a variable then the content of* v *is* moved back[3] *into* u *before control returns to the caller.*

**Rule 4** *The value of an unshareable formal parameter* v *cannot be changed by the procedure* r *in which it is defined, or by any procedure (recursively) called by it. This entails the following:*

1. *No assignment into* v *is allowed. (Actually, this constraint is already imposed by Eiffel itself, on* all *formal parameters).*

2. v *cannot be assigned to any variable. (If permitted, such an assignment would nullify* v *due to Rule 2.)*

3. v *cannot be* recycled *(Recycling of unshareable objects is defined in Section 4.5.)*

Since, by Rule 4, the formal parameter cannot be modified, it follows from Rule 3 that unshareable parameters lent by a procedure call would be returned when this procedure exits.

Note, however, that in spite of these two rules, an unshareable parameter would be nullified when the same u-variable is used more than once in a given procedure call. To see this, consider a procedure call `x.p(u,u)`, and let `v1` and `v2` be the two corresponding formal parameters. Under Rule 3, the passing of u to `v1` (if this is the first argument to be bound) would nullify u, and it is this null value which will pass to `v2`. If the formal parameters are moved back to the actual ones in the same order then it is easy to see that u will end up being nullified. This is one reason why such aliasing is not recommended, another reason is that the order of binding determines which of the formal parameters will be nullified, and thus, eventually, the result of

[3] This is an adaptation of the so called *value-result* parameter passing [3]

the computation. It should be pointed out, though that even multiple use of the same actual parameter would not cause any u-pointer to be duplicated.

### 4.3.1 Unshareable Parameters that can be Consumed

As we have already pointed out, one needs sometimes a routines which does consume some of its unshareable parameters. To provide for such consumption in a predictable fashion we allow for an unshareable formal parameter to be explicitly declared as `consumable`. Such parameters are subject to the following rule:

**Rule 5** *Let an unshareable formal parameter* `v` *of a procedure* `r` *be declared as* `consumable`*, then:*

1. *Procedure* `r` *is permitted to assign* `v` *into some other variables, thus nullifying* `v`*. (This relaxes point (2) of Rule 4.)*
2. *A* non-consumable *formal parameter of some routine* `r1`*, i.e., a formal unshareable parameter not declared as consumable, cannot be bound to a consumable formal parameter such as* `v`*.*

Thus, if a routine `push(u)` defined for a stack `s` of unshareable objects declares its formal argument as `consumable`, then the call `s.push(u)` may cause the pointer in `u` to be moved permanently into the stack, and `u` itself to be nullified, as is appropriate in this case (see example in Section 4.6).

### 4.3.2 Operations on Unshareable Objects

Finally, we must deal with operations of the form

```
u.m(...),
```

where `u` is an unshareable variable. The problem here is that `u` must be considered a parameter to its own method `m`. In Eiffel, in particular, `u` is bound by this operation to the implicitly defined local variable `current`[4] of method `m`. In order to preserve the uniqueness of the pointer in `u`, and prevent this pointer from being consumed by `m`, we need this binding to be *by lending*. But if `m` happens to violate Rule 4, in particular by assigning `current` to some other variable, then `u` will be consumed by this operation. This disconcerting prospect can be prevented by employing the following rule, (which, like all the rules of Eiffel*, can be checked at compile time):

[4]The equivalents in other languages have names such as "self" or "this".

**Rule 6** *Variables of a class* `C` *can be declared as u-variable* only if *all the methods defined for this class treat their implicitly defined local variable* `current` *as non-consumable formal parameter, satisfying the constraints of Rule 4.*

This constraint on the the classes whose variables can be declared as unshareable is not as restrictive as it may seem, for two reasons: First, the conditions imposed by this rule on the use of variable `current` are almost always satisfied in normal use. For example, an analysis of the official Eiffel library indicates that less than 2% of its classes violate this rule, and the unalysis of three fairly large randomly chosen applications programs revealed *no* such violations. Second, even if some method of a given class `C` does not satisfy Rule 6, it is often possible to define a class `C1` that inherits from `C`, redefining the offending methods in it, so that `C1` would satisfy our rule and can thus be used as a basis for unshareable variables.

### 4.4 Miscellaneous Rules

We describe here the rest of the rules that support unshareable objects in Eiffel*. These rules tend to be more specific to the Eiffel language, and of a somewhat lesser general import than those considered above. The statement of each rule is preceded by its motivation.

First, most languages provide some means for copying entire objects. (In Eiffel this can be done by means of explicit copy routines such as `copy` and `clone`, and by the assignment of *expanded* objects, which are used infrequently in this language.) Such a copy is problematic if an object being copied contains unshareable attributes. The copying of objects must, therefore, be subjected to the following rule (stated in very general terms):

**Rule 7** *The copying of a complete object must not be allowed to copy any unshareable attribute of it. Such attributes must be either moved, according to Rule 2, or not transferred at all by the copy routine. (Another possibility is to completely disallow any copying of objects with unshareable attributes.)*

Second, we confront the following problem[5]: if an object `x` has an *exported* unshareable attribute `u`, then due to Rule 2, the assignment statement

```
v := x.u;
```

[5]This problem has been pointed out by Partha Pal

would consume the u attribute of x. But this would violate one of the basic properties of encapsulation in Eiffel, namely that it is not possible to change the value of an attribute of an object directly from the outside. To prevent this violation we impose the following rule:

**Rule 8** *An unshareable-attribute of a class cannot be exported.*

Of course, this does not prevent an object from "voluntarily" giving up one of its private unshareable attributes, returning it as a result of one of its methods.

Finally, we must impose the following constraint on *once functions*, which is an unusual Eiffel device designed to support globally accessible objects:

**Rule 9** *The result of a* once function *cannot be declared as unshareable.*

The reason for this rule is that a once function in Eiffel returns the same result every time it is called. This result, then, is not unique, and thus cannot be unshareable.

### 4.5 Recycling of Unshareable Objects

The Eiffel language provides no explicit means for the deallocation of dynamic objects. Because such means would be unsafe, due to possible *dangling reference*, and because they are considered unnecessary in a language with garbage collection. The deallocation of unshareable objects, however, is quite safe, and, as we shall see, it can be very helpful even in the presence of garbage collection. The following rule introduces an appropriate deallocation method, `recycle`, for unshareable objects. (Note that this rule, and the following one, are not required for the support of the concept unshareable objects itself but they can help making the most out of such objects.)

**Rule 10 (the `recycle` method)** *Let the class* `ANY` *have a method* `recycle`*, which can be applied only to u-variables. Method* `recycle` *does nothing when applied to a void variable, but when applied to a non-void unshareable variable* `u`*, it operates as follows:*

1. *It applies* `recycle` *(recursively) to all unshareable attributes of* `u`*;*
2. *It deallocates the object pointed to by* `u`*, and then nullifies variable* `u` *itself.*

Note that if `u` is the head of a tree then `u.recycle` would recycle the entire tree. (This procedure terminates because pointers to unshareable objects cannot form a cycle.)

Recycling of unshareable objects can be done in two ways: *manually*, whenever one decides that an object is not needed anymore, or *automatically*, whenever it is evident that an object *cannot be used* anymore. Such automatic recycling is established by the following rule.

**Rule 11 (automatic recycling)** *The* `recycle` *method introduced in Rule 10 is applied automatically, as follows:*

1. *Before a procedure exits all unshareable objects addressed by its local variables are recycled (i.e., the method* `recycle` *is applied to them.)*

2. *Before an assignment* `u := v` *is carried out,* `u` *is recycled.*

The implications of these rules are discussed briefly in Section 6.

### 4.6 On the Use of Unshareable Objects

Being used, as we are, to the traditional transfer-by-copy in programming, the use of unshareable objects which *move* from one place to another may seem strange, and even disconcerting. But such uneasiness is not justified, particularly not for data structures that are meant to model physical objects, or that are meant to exhibit such properties of physical objects as the *containment relation* between them. In fact, it is quite possible that in many applications unshareable objects would end up being the norm, which would dramatically reduce the cost of storage management, as is argued in Section 6.

To illustrate the use of unshareable objects in programming we introduce in Figure 2 a code of a class `U_STACK` for unshareable objects of certain class `U` (we assume that class `U` itself is declared as unshareable, which make all the instances of U unshareable by default). We use here a list of conventional (sharable) nodes (defined by a class `NODE`) to represent a stack, but each of these nodes points to an unshareable object. (This is done in order to show how conventional and unshareable variables can be mixed in a single class; in fact, the nodes of a stack can also be made unshareable). This is a pretty standard implementation of a stack, but it has some unusual aspects that are discussed below.

First, the method `push` actually consumes its argument, which it can do because the formal parameter of `push` is defined as `consumable`. For example, consider an object `x` that has an unshareable attribute `u1` containing a

pointer `p` to an object of class `U`, and an attribute `s` that points to one of our stacks. The statement

```
s.push(u1)
```

carried out by `x` would move pointer `p` into the stack, nullifying `u1`. Object `x` can get pointer `p` back by means of a statement such as:

```
u2 := s.pop;
```

which would move `p` from the stack into variable `u2`.

Second, note this stack does not have any `pop` method, because it is inherently impossible to return the unshareable top of this stack without removing it from the stack. (Although it is possible to approximate the conventional `pop` method by producing a copy of the object addressed by the top of the stack, and returning a pointer for this copy.)

# 5 Related Work

This work bears significant similarities to two recent efforts. Both support unshareable objects (using different terminologies) and for some of the same reasons the motivated this work. But there are deficiencies in both of these proposals, particularly for object-oriented programming (which, in fairness, was not the context in which these proposal were made.)

The first of these efforts is by Harms and Weide [4], who may have been the first to challenge the conventional use of copying as the primary mechanism for transferring data in programming. They proposed to replace all such transfers (i.e., assignment and parameter passing) with *swaps*, which would make *all* dynamic objects unshareable.

One problem with this proposal is that swapping, as a mechanism for the transfer of data, is inconsistent with the polymorphic, strongly typed, object oriented languages. This is because in such languages the type constraints on assignments are *antisymmetric*, and thus incompatible with the symmetric swap. This problem can be demonstrated as follows: Let class `C1` be a proper superclass of `C2`, and let `v1` and `v2` be variables of classes `C1` and `C2`, respectively. Now consider the assignment statement

```
v1 := v2;
```

Such statements are allowed in Eiffel, and are very important to OO languages in general, because they provide for polymorphism. But the swapping

```
      -- The parts of this code that deals specifically with
      -- its unshareable aspects are commented.
        -- We assume that class U is declared as unshareable,
        -- which make all the instances of U unshareable by default.
class NODE
  feature {NONE}
    item:U
                      -- Note that item is defined as a private feature;
                      -- it must be due to Rule 8
    next:NODE
  feature
    set_item(u:U consumable ) is  -- u is a consumable argument
      do item := u    -- this is a move which consumes u
      end;
    set_next(n:NODE) is
      do next := n
      end;
    get_item:U is                 -- returns an unshareable result
      do result := item           -- this is a move which consumes item
      end;
end -- class NODE
```

---

```
class STACK
  feature {NONE}
    last:NODE
  feature
    push(u:U consumable) is       -- u is a consumable argument
      local n:node
      do
        !!n
        n.set_item(u) -- u is thus consumed by node n
        n.set_next(last)
        last := n
      end;
    pop:U is          -- This function returns an unshareable pointer.
      local n:node
      do
        result := last.get_item   -- the top item is thus moved into result
        last := last.next
      end;
end -- class STACK
```

Figure 2: Stack of unshareable objects

paradigm would replace this statement with a swap of values which, in particular, will place the value of `v1` into `v2`, violating the requirement that a variable should not hold instances of its superclasses [5].

Another problem with the scheme proposed by Harms and Weide is that it fails to protect unshareable parameters from being consumed by the procedure they are submitted to. It should be pointed out that when dealing with specification, which is the context of the proposal by Harms and Weide, one could, perhaps, be satisfied by having each procedure specify whether or not it consumes any of its parameters. But, when dealing with the programming of large systems one cannot rely entirely on the specifications of all its components, because we usually have no assurance that the specifications are really satisfied. And the very possibility that an unshareable parameter can be unexpectedly consumed by a procedure may be devastating in this context. This particular difficulty is even more serious in Baker's proposal discussed below.

The second related work has been recently reported in a very interesting paper by Baker [1]. Baker introduces a concept of *linear objects*, which are like our unshareable objects, but are handled differently. Linear objects are addressed by what Baker calls "use-once" variables, because every use of such a variable consumes its value. That is, if `u` is a use-once variable, then every procedure call `p(...,u,...)`, and every operation `u.m`, nullifies it. This is a serious drawback, which would make programming with unshareable object very difficult and very unsafe, particularly in the context of an object oriented language. Baker himself states that *"The acceptance by a function of a linear argument object places a great responsibility on the function..."*. Because, if the argument of a function is to be retained by the caller, it must be returned to him as a value of this function. Baker admits that this would make writing programs syntactically complex, because functions may have to have several return values; and he proposes a graphical language as a solution. But what is perhaps worse about this scheme is that the failure of a function to return some of its linear (unshareable) arguments may cause very grave consequences to the internal state of its caller, by having some of its private components consumed.

# 6 Applications and Implications of Unshareable Objects

We have demonstrated the usefulness of unshareable objects as strictly hidden components of other objects, thus resolving a serious problem with en-

capsulation in conventional object-oriented programming. Another important application of unshareable objects is discussed in [7], where we show how such objects can be used to implement *tokens* — objects that, like the *capabilities* of operating systems, represent certain authority. Such unshareable tokens can be utilized, in particular, for the control of sharing in software systems, when sharing is desirable. Besides such specific situations where unshareability is clearly required, we believe that unshareable objects are usable in a broad range of applications where no sharing is intended, and that such use would tend to make systems simpler and more reliable.

In addition to the salutary effect of unshareable objects on our ability to reason about systems, a massive use of such objects should also have a significant beneficial effect on the safety and efficiency of storage management. This effect is due to the fact that unshareable objects can be safely *recycled*, as described in Section 4.5. The precise nature of this effect depends on whether or not the language in question provides garbage collection.

In a language with garbage collection (like Eiffel) the manual and automatic recycling of unshareable objects should reduce the frequency of invocation of the expensive garbage collection procedure, thus making storage management more efficient. In a language without garbage collection (like C++) the use of unshareable objects should have two beneficial effects on storage management: First, the conventional unsafe deallocation of dynamic objects would be replaced by the safe explicit recycling. Second[6], the automatic recycling of manifestly unreachable unshareable objects, as defined by Rule 11, should reduce the amount of *memory leakage* in the system, i.e., the number of allocated objects that have *no* pointers leading to them, and are therefore lost for the program.

*Acknowledgment*: I would like to thank Alex Borgida, Rock Howard, Yaron Minsky, Partha Pal and Yossi Stein for many useful conversation during the writing of this paper.

## References

[1] H.G. Baker. 'use-once variables and linear objects — storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, January 1995.

[6] I owe the last observation to Yaron Minsky.

[2] L. Cardelli, J. Dinahue, L. Glassman, M. Kalsow Jordan, B., and G. Nelson. Modula-3 report (revised). Technical Report 52, Digital System Research Center, November 1989.

[3] Fischer A. E. and Grodzinsky F. S. *An Anatomy of Programming Language*. Prentice Hall, 1993.

[4] Dougles E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, pages 424–434, May 1991.

[5] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1987.

[6] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[7] N.H. Minsky. On the use of tokens in programming. Technical report, Rutgers University, LCSR, February 1995. (To be finished).

[8] R. Sethi. *Programming Languages, Concepts and Constructions*. Addison Wesley, 1989.

# Contents