Towards Alias-Free Pointers

Naftaly H. Minsky*

Rutgers University, New Brusnwick NJ 08903 USA
Tel: (908) 445-2085; e-mail: minsky@cs.rutgers.edu

Abstract. This paper argues that pointer-induced aliasing can be avoided in many cases by means of a concept of *unique pointer*. The use of such pointers is expected to fortify the concept of encapsulation, to make systems easier to reason about, to provide better control over the interaction between threads, and to make storage management safer and more efficient. We show that unique pointers can be implemented by means of few minor and virtually costless modifications in conventional OO languages, such as Eiffel or C++; and that they can be used conveniently in a broad range of algorithms and data structures.

Key Words and Phrases: pointer-induced aliasing, hiding, encapsulation, programming with threads, storage management.

* 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 mixed blessing in imperative programming. A blessing, because dynamic objects have some very useful properties, such as *indefinite lifetime*, *indefinite scope*, *efficient transferability*, and the ability to be *shared* by multiple pointers to a single object. But the shareability of dynamic objects via pointers dispersed throughout a system is very problematic. It allows for aliases to exist for a given dynamic object, anywhere in the system, making it hard to to reason about this object; and it undermines the principles of *hiding*, and of *encapsulation*, the very foundations of object-oriented programming. The virtually uncontrollable dispersal of pointers also makes storage management more hazardous and costly.

In this paper we argue that pointer-induced aliasing is largely a self inflicted wound, caused by the almost universal practice in programming to transfer information *by copy*. As a remedy for this defect we introduce a concept *unshareable objects*, and the companion concept of *unique pointers*, which can be *moved* from one place to another, but which cannot be copied. We argue that unshareable objects can be employed conveniently in many, if not most, situations where dynamic objects are being used, and without incurring their pitfalls. And we show that it takes no more than minor, and virtually costless, modifications to a typical imperative programming language to support such objects.

For the sake of specificity we couch our discussion in terms of the object-oriented language Eiffel [7]. 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. (A close approximation to unique pointers under C++ has been constructed.) The rest of this paper is organized as follows: The pitfalls of conventional pointers are discussed in Section 2; in Section 3 we describe a simple variant of Eiffel that supports unique pointers; the use and applications of such pointers are discussed in Section 4; and some related works are discussed in Section 5.

2 The Pitfalls of Dynamic Objects

Under conventional programming languages, dynamic objects have several pitfalls, some are better known than others. We start by showing that dynamic objects are very difficult to hide in any specific locale, due to the virtually uncontrollable dispersal of pointers to such objects. We then briefly discuss the difficulties caused by conventional dynamic objects to *storage management*, to *encapsulation*, and to programming with *threads*.

2.1 The Difficulty in Hiding Dynamic Objects

Although the concept of hiding in software is well known [9] — and is widely considered the bedrock of modularization and of encapsulation — it is a somewhat slippery concept, that may have several definitions reflecting different concerns. The following is one such definition, whose full significance will become clear in due course.

Definition 1 (a 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.)*

Note that although this 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.)

Now, if a component `c` of an object `x` is physically *contained* in it, as illustrated in part (a) of Figure 1, then the condition of Definition 1 can be readily established by the scope rules of the language.[2] But if `c` is a dynamic object, addressed via a reference variable `p_c` contained in `x` then, the scope rules are not

[2] 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 C++, or in procedures of Eiffel. We ignore the effect of such unsafe features in this paper.

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.

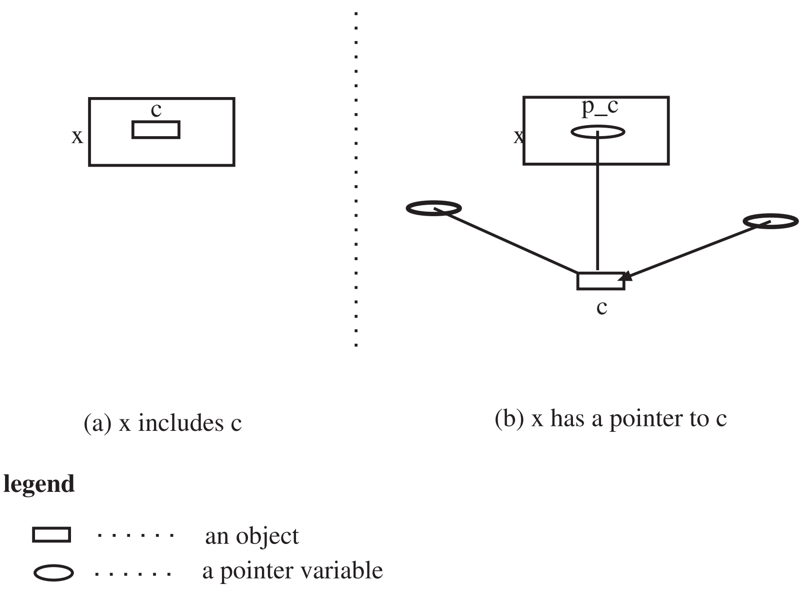

Fig. 1. The Difficulty in Hiding Dynamic Components

Moreover, under conventional languages, it is virtually impossible to control the dispersal of pointers for a given object throughout a system, or to figure out the extent of such dispersal [5]. This is due to the fact that information is generally transferred from one place in a system to another *by copy* — the copy of pointers when dealing with dynamic objects. This, in particular, is the case in Eiffel for the *assignment statement*

```
u := v;
```

which (when `v` is a *reference variable*) copies the pointer in `v` into `u`, leaving `v` intact — thus creating a duplicate of this pointer. Therefore, if object `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. Moreover, even if `x` itself is the original creator of `c`, there is very little it can do to prevent the leakage of pointers to `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 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`.

2.2 The Adverse effects of Dynamic Objects on Storage Management

In a language without garbage collection, where dynamic objects need to be deallocated explicitly, the uncontrollable dispersal of pointers causes two kinds of dire phenomena. First, when an object is deallocated, any surviving pointer to it, left anywhere is the system, becomes a *dangling reference*, which may cause serious errors that are notoriously difficult to debug. Second, partially due to the fear from dangling reference one is often reluctant to deallocate an object even when there is no apparent need for it. This contributes to *memory leaks*, which depletes the memory available to the program.

It is because of the specter of dangling reference that languages with garbage collection do not provide for explicit deallocation of objects. But, as we shall see later, if such deallocation can be made safe it can be very useful in such languages, by reducing the amount of garbage collection needed.

2.3 The Conflict Between Encapsulation and Pointers

Encapsulation is based on the *hiding* of the constituent parts of the state of an object from anything but its own program. Such hiding is supposed to have two distinct consequences, which are critical for large systems: First, it should provide objects with *implementation transparency*; i.e., the ability to change the internal representation of the state of an object (together with its program) without having to change anything in the rest of the system. Second, encapsulation is supposed to enable us to endow an object with what is sometimes called *invariants* (or *class invariants*). These are properties that *"hold whenever* *control is not in the object"* (Sethi ([10]), and which are completely independent of the rest of the system.

Now, while implementation transparency can be achieved by "weak hiding," via scope rules, invariant properties require the stronger kind of hiding of Definition 1. Indeed, our ability to establish properties of an object which are independent of the rest of the system, is clearly undermined if this object has dynamic components, which may be accessible to any number of other objects.

This is a serious problem because invariant properties are essential for meaningful encapsulation, and for abstract data types. Yet, although this problem with encapsulation is not unknown (see [6] page 159, in particular) it is rarely discussed in the literature, and has not been satisfactorily resolved so far.

2.4 A Difficulty with Threads

The dispersal of pointers also has an adverse effect on *programming with threads* [11]. Suppose that `x` is built as a *monitor*, meaning that only one thread can gain access to the internals of `x` at any given moment in time. This is supposed to prevent *race conditions* between processes when they manipulate `x`.

Unfortunately, if the components of `x` are dynamic objects, then the mutual exclusion with respect to `x` does not prevent race conditions. This is illustrated in Figure 2, where the component `c` of `x` is accessed concurrently by two threads: `T1`, which gained exclusive access to `x`, and `T2`, which operates concurrently outside of `x`, but which may operate on component `c` of `x` through one of the pointers to it dispersed in the system.

3 Unshareable Objects & Unique Pointers

For situations where the dispersal of pointers is undesirable, we introduce the following concepts:

Definition 2 *A dynamic object is called* ***unshareable****, or a* u-object*, if there* can *be only one pointer in the system leading to it. A pointer to a u-object is called a* u-pointer*, in part because it is guaranteed to be* ***unique****.*

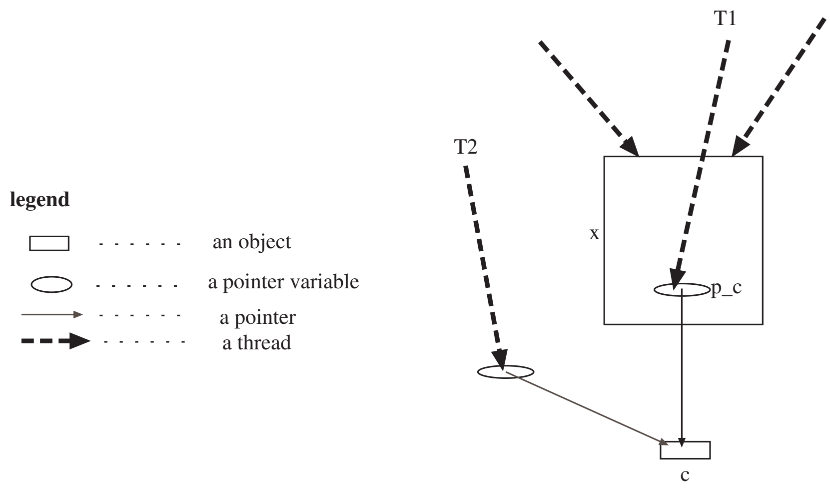

Fig. 2. Race Condition Between Threads

We show in this section how these concepts can be supported, and made usable, under the object-oriented language Eiffel. Technically, we describe a variant of Eiffel, obtained by a small set of minor modifications (defined by a set of rules) 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, if necessary. (In certain languages, such as C++, some of these rules can be established without any changes in the language itself.)

The implementation proposed here for u-objects rests, in part, 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*. In addition, some constraints are required on the treatment of formal parameters of procedures. The compile-time cost and the run-time overhead required to establish the rules advocated here for Eiffel* turn out to be quite negligible. Moreover, these rules impose no constraint on anything not involving u-objects.

A disclaimer is in order here: The assurance provided by Eiffel* that pointers claimed to be u-pointers are in fact unique is not absolute. It provides about the same level of the certainty that Eiffel provides for type correctness and for its scope rules. All such integrity conditions 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 3.

The rest of this section is organized as follows: We start with a concept of `u-variables`. These are variables declared as `unique`, each of which contains either a u-pointer or void. This is followed by rules that define the treatment of u-variables by the assignment statements, by parameter passing, and by some other constructs of Eiffel. Finally, in Section 3.5 we introduce means for safe and efficient recycling of u-objects.

Two comments about terminology, before we start: First, we use the term "variable" for what is called an "entity" in Eiffel, which is a name used in the program text to refer to values, which may, at run time, be associated with objects. Second, Our term "unique" in this paper should not be confused with an integer entity declared as "unique" in Eiffel.

3.1 u-variables

Eiffel* allows variables of any class to be declared as *unique*, subject only to a restriction imposed by Rule 4, introduced later. Such variables, which are guarantees to contain either u-pointers or void, 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; we will refer to such variables as u-attributes, u-parameters, etc. We also provide for a class to be declared as unique, which means that all variable declared to be of this class are u-variables. The first rule of Eiffel* is as follows:

Rule 1 *No transfer of pointers from regular variables into u-variables is allowed.*

In particular, this rule prohibits the assignment of a regular variable into a u-variable, and the passing of a regular actual parameter into a formal parameter declared as unique. The reason for this prohibition is, of course, that a regular variable may contain a non-unique pointer.

3.2 Assignment of u-variables

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 causes pointers to u-objects to be *moved* by an assignment, instead of being copied.

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

In other words, if `u` is a u-variable then its pointer *moves* into `v`, leaving void in its wake. Note that if the right hand side of an assignment is a function call that returns 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. (The assignment statement is also subject to the optional Rule 9, which deals with the deallocation of unusable u-objects.)

Note that instead of changing the semantics of the conventional assignment operator, one may prefer to prohibit its use on u-variables, adding a new *move* operator to replace it. This is largely a matter of taste.

3.3 The Treatment of Formal Unshareable Parameters

If parameter passing is done *by reference*, as is the case in Eiffel, it causes no duplication of pointers. But it presents another kind of difficulty: a u-variable `u` used in a call `p(u)` may be *nullified* by this call. This would happen, in particular, if the corresponding formal parameter `v` is assigned to any other variable. By Rule 2, this would be a move-assignment that nullifies `v`, as well as `u`. In a sense, procedure `p` *consumes* the u-pointer given to it in `u`.

Although such consumption of an actual argument is sometimes required, it is generally undesirable. It is required, for example, in the following situation: Let object `x` have a u-component `u`, and suppose that `x` performs the operation

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

where `s` is a stack. The pointer in `u` *must* be consumed by this operation, so that it can be stored in the stack, since it points to a u-object. Object `x` should "expect" `u` to be nullified by this call. On the other hand, suppose, that `x` performs the operation

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

which is supposed to display `u` on the user-terminal `t`. It would be quite unreasonable for this call to consume `u`, yet this is precisely what would happen if procedure `display` assigns its formal parameter to anything.

The very concept of unshareable objects would be quite untenable without the means for ensuring that an actual parameter cannot be consumed, when consumption is not intended. We, therefore, require a formal u-parameter to be declared as either `consumable` or `non-consumable` (we take the latter, which is likely to be the more common one, as the default). The treatment of non-consumable formal parameters is subject to the following constraints:

Rule 3 *The value of a formal u-parameter* `v` *declared as* non-consumable *cannot be changed. This entails the following constraints on the treatment of such parameters:*

1. *No assignment into* `v` *is allowed. (Actually, in Eiffel this constraint is already imposed on* all *formal parameters).*
2. `v` *cannot be assigned to any variable.*
3. `v` *cannot be used as an actual argument in a procedure call if the corresponding formal parameter is declared as consumable.*

There is an important special case of parameter passing in OO programming which requires special attention here. This is an operation of the form

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

where `u` is a u-variable of some class `C`, and `m` is one of the methods of `C`. The problem here is that `u` itself might be nullified by this operation — quite a disconcerting prospect. The reason for this is that `u` must be considered a parameter to its own method `m`. In Eiffel, in particular, it is bound to the implicitly defined local variable `Current`[3] of method `m`. Now, if `m` happens to assign `Current`

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

to some other variable, then `u` will be consumed by this operation. Such consumption would not occur if `m` satisfies the constraints of Rule 3 with respect to variable `Current`. This is ensured by the following rule[4], (which, like all the rules of Eiffel*, can be checked at compile time):

Rule 4 *Variables of a given 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 3.*

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 of ISE (their EiffelBase) indicates that less than 2% of its classes violate this rule, and an analysis of three (fairly randomly chosen) applications programs revealed *no* such violations. Second, even if some method of a given class `C` does not satisfy Rule 4, 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 u-variables.

3.4 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 u-

[4] We point out, in response to a question by Bertrand Meyer, that the call `u.m(u)`, where u is a u-variable and the argument of `m` is `consumable`, would cause `u` to be consumed, after the call is carried out. We dot not find this consequence, of this rare construct, to be particularly distressing.

attributes. The copying of objects must, therefore, be subjected to the following rule (stated in very general terms):

Rule 5 *The copying of a complete object must not be allowed to copy any u-attributes 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* u-attribute `u`, then due to Rule 2, the assignment statement

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

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 6 *A u-attribute of a class cannot be exported.*

Of course, this does not prevent an object from "voluntarily" giving up one of its private u-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 7 *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.

3.5 Recycling of Unshareable Objects

The Eiffel language provides no explicit means for the deallocation of dynamic objects. This is, because such means would be unsafe due to possible *dangling reference*, and because they are considered unnecessary in a language with garbage

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

collection. The deallocation of u-objects, however, is quite safe, and, as we shall see, can be very helpful even in the presence of garbage collection. The following rule introduces an appropriate deallocation method, `recycle`, for u-objects[6].

Rule 8 (the `recycle` method) *Let there be a method* `recycle` *that can be applied to any u-variable* `u` *which is not an unconsumable argument of a procedure. Method* `recycle` *does nothing when* `u` *is void, and operates as follows, otherwise:*

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

Note that `recycle` terminates because pointers to u-objects cannot form a cycle.

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

Rule 9 (automatic recycling) *The* `recycle` *method introduced in Rule 8 is applied automatically, as follows:*

1. *Before a procedure exits all u-objects addressed by its local variables are recycled (i.e., the method* `recycle` *is applied to them.)*
2. *When an object is collected, during garbage collections, all its unshareable components are recycled.*
3. *Before an assignment* `u := v` *is carried out,* `u` *is recycled.*

(Note that automatic deallocation can be established easily in C++, but it would not be generally safe unless it is applied to u-variables.) The implications of recycling for storage management are discussed briefly in Section 4.4.

4 On the Use and Applications of Unshareable Objects

Being used, as we are, to the traditional transfer-by-copy in programming, the use of u-objects requires a change of viewpoint — one should think about them

[6] This rule and the following one are not required for the support of unshareable objects, but they can help making the most of them.

as things that *move* from one place to another, just like the physical objects we manipulate in daily life. As an example of this difference, note that a stack designed to maintain u-objects cannot have the traditional `top` method, which returns a pointer to the top of the stack, without removing it. This is inherently impossible when this top is unshareable.[7] In spite of such unfamiliar aspects of unshareable objects, we will show in this section that they can be used quite conveniently in many applications, and can even be shared via intermediate "handle" objects, when such sharing is required.

We start this section with a very simple programming example, introducing regular handle-objects that can be used to effectively share u-objects. In Section 4.2 we discuss applications for which u-objects are naturally suitable. In Section 4.3 we show that u-objects can be maintained in arbitrary data-structure, which makes them usable in a broad range of applications. Finally, in Section 4.4 we discuss the implication of u-objects to storage management.

4.1 Shareable Handles for U-Objects

Suppose that the instances of a given class `C` are meant to be sometimes hidden in some other objects, and be sometimes easily accessible to many parts of the system. These seemingly contradictory usages can be supported by (a) making all instances of `C` unshareable, so they can be truly hidden, when hiding is called for; and (b) making them accessible through *shareable handles*, when wider accessibility is called for.

To see in detail how this can be done, consider the class `HANDLE` defined in Figure 3 (assuming, for simplicity, that class `C` itself is declared as unshareable). Every handle has an u-attribute of class `C`, called `body`; and there are two methods applicable to it: the method `install(b)`, which installs `b` as the body of the handle, and `remove`, which removes the body of the handle, returning it as its value.

Consider, now, a regular (i.e., shareable) handle `h` that has an object `p` as its body. Although `p` itself is unsahreable, it can be accesses through `h` by anybody

[7] But one can approximate the conventional `top` method, if such is desired, by producing a copy of the top of the stack and returning a pointer for this copy.

```
class HANDLE
  feature
    body:C            -- This is a u-variable, because class C is unshareable
    install(b:C consumable) is    -- b is a consumable argument
      do body := b    -- this is a move that consumes the actual argument
      end;
    remove:C is       -- the result of this function is unshareable
      do result := body           the body is consumed by this method
      end;
end -- class NODE
```

Fig. 3. A handle for an u-object

that has a pointer to it, like object `x` in Figure 4. In particular, if `m` is one of the methods of `p` then `x` can perform the operation

```
h.body.m,
```

thus applying `m` to `p`; and so can other objects that have access to `h`. This arrangement allows `p` to be shared, in spite of its unshareable status; but it also allows one to hide `p` at will. In particular, a statement

```
v:=h1.remove
```

carried out by `x` (see Figure 4), *moves* `p` from `h` into `v`, thus hiding this object inside `x`, regardless of who shared it before via the handle `h`. Finally, `x` may at any time return `p` to handle `h` by means of operation

```
h1.install(v)
```

which moves `p` back into `h`, making `p` widely accessible again.

4.2 Natural Applications of Unshareable Objects

Perhaps the most natural and important application of u-objects is as a means for fortifying encapsulation. Such fortification is called for when: (a) an object

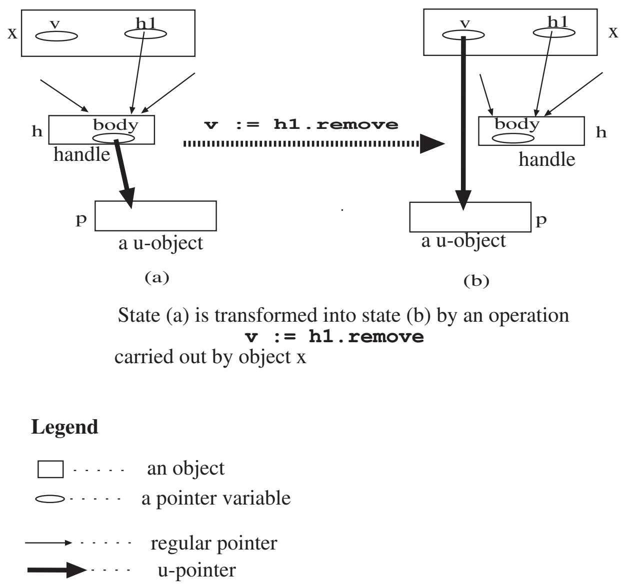

Fig. 4. The sharing of a u-object via a handle

has a varying number of components, or (b) the components of an object have an indefinite lifetime, or (c) when components are being moved dynamically from one object to another. If any of these conditions are satisfied one needs to have the components of an object allocated on the heap, which, as argued in Section 2, makes them hard to hide effectively, unless they are made unshareable. Making the component parts of an object unshareable would facilitate, in particular, the construction of reliable class invariants, and the prevention of race conditions between threads.

Another natural application of u-objects is discussed in [8], 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 such as object-oriented databases.

4.3 General Programming with Unshareable Objects

Besides the above natural applications of u-objects, such objects can be used effectively in a broad range of applications where the ability to prevent aliasing is important. Indeed, there is no serious limitation to the applicability of u-objects. We already saw that u-objects can be effectively shared via handle objects. One can, therefore, build arbitrary data structure involving u-objects, although such objects cannot be used *directly* as nodes in many kinds of list structures, such as in doubly linked lists. Indeed, employing a simple generalization of the handles of the previous section, u-objects can be stored in any graph, with one level of indirection, as is illustrated in Figure 5 for doubly linked lists. Although such indirection involves a certain amount of overhead, it is very common in data structures anyway, and should not be seen as a serious limitation on the use of u-objects.

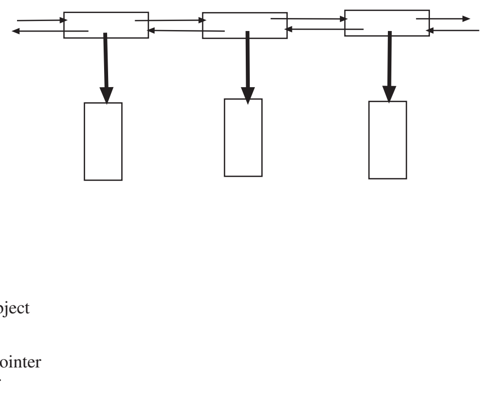

Fig. 5. A doubly linked list of u-objects

Of course, not all dynamic objects can be made unshareable. At the very least, we need regular objects to serve as handles for unshareable ones, allowing for controlled sharing of u-objects, and for their incorporation in complex data structures.

Finally, it should also be pointed out that u-objects can contain pointers to regular objects, and that regular objects can point to u-objects (which is the case with handles, for example). The one thing that should not be allowed is to have a pointer into an internal component of a u-object. No such pointers are possible in Eiffel, but they can be generated in C++ by means of the & operator, which should be prohibited in any implementation of u-objects in this language.

4.4 The Effect of Unshareability on Storage Management

Unshareable objects should have a significant beneficial impact on the safety and efficiency of storage management, particularly if they are used massively. This is due to the fact that u-objects can be *recycled* safely, as described in Section 3.5. The precise nature of this impact depends on whether or not the language in question provides garbage collection.

In a language without garbage collection (like C++) the use of u-objects should have two beneficial effects: First, the conventional unsafe deallocation of dynamic objects would be replaced by the safe explicit recycling. Second[8], the automatic recycling of manifestly unreachable u-objects, as defined by Rule 9, 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 to the program.

In a language with garbage collection (like Eiffel) the manual and automatic recycling of u-objects should reduce the frequency of invocation of the expensive garbage collection procedure, thus making storage management more efficient. This effect would be particularly strong if u-objects are used for all but the handle-objects required for complex data structure. This is because in this case mostly the handle objects would be subject to garbage collection, and since such objects are likely to have several standard small sizes, which are easier to manage.

5 Related Work

This work bears significant similarities to two recent efforts. Both support objects that satisfy our Definition 2 and for some of the same reasons that 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.)

One 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.,

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

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 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 [6]. Another problem with the scheme proposed by Harms and Weide is that it fails to protect u-parameters from being consumed by the procedure they are submitted to. This particular difficulty is even more serious in Baker's proposal discussed next.

The second work has been recently reported in a paper[9] by Baker [1]. Baker introduces a concept of *use-once* variable, which point to what he calls *linear object*. His concept of linear object, which was inspired by Girard's *linear logic* [3], is equivalent to our u-object. But we find the manner these objects are handled, via use-once variables, problematic. The term "use-once" variables, indicates that *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 u-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

[9] This paper also contains a fairly extensive review of previous related works

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 Conclusion

We have argued in this paper that the serious problem of pointer-induced aliasing is largely a self inflicted wound, caused by the almost universal practice in programming to transfer information *by copy*. We have shown that it takes only few minor, and virtually costless, modifications of a typical programming language (involving, in part, the transfer of pointers) to ensure that certain variables contain unique pointers. And we have argued that the use of such variables are likely to have a salutary effect on our ability to reason about large systems, and on the safety and efficiency of storage management.

Although the complete implementation of u-pointers and u-objects generally requires some changes to the definition of a language, of the kind described here for Eiffel, we have constructed a good approximation for u-objects in C++, which requires no changes in the language itself. This construction, which has been carried out by Yu-min Liang from Rutgers University, only approximates u-objects in that it relies on a program to satisfy certain simple constraints, such as that the & operator is never applied to objects designated as unshareable. This construction will be described elsewhere.

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. I am also indebted to Bertrand Meyer for his many useful comments when reviewing this paper.

References

1. H.G. Baker. Use-once variables and linear objects — storage management, reflec-

tion and multi-threading. *ACM SIGPLAN Notices*, January 1995.

2. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, and G. Nelson. Modula-3 report (revised). Technical Report 52, Digital System Research Center, November 1989.
3. J. Y. Girard. Linear logic. *Theoretical Computer Science*, pages 1–102, 1987.
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. W Landi. Undecidebility of static analysis. *Lett. Program. Lang. Syst.*, 1(4), December 1992.
6. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1987.
7. B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
8. N.H. Minsky. On the use of tokens in programming. Technical report, Rutgers University, LCSR, October 1995.
9. D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), December 1972.
10. R. Sethi. *Programming Languages, Concepts and Constructions*. Addison Wesley, 1989.
11. A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

Table of Contents

This article was processed using the LaTeX macro package with LLNCS style