

The Programming Language Gilgul

Pascal Costanza

University of Bonn, Institute of Computer Science III
Römerstr. 164, D-53117 Bonn, Germany

costanza@cs.uni-bonn.de, <http://www.pascalcostanza.de>

Keywords: dynamic object replacement, object identity, language design

Classification: within one year of thesis completion

1 Introduction

Unanticipated Software Adaptation Software requirements are in a constant flux. Some changes in requirements can be anticipated by software developers, but unanticipated changes of requirements occur repeatedly in practice, and they cannot be prepared for by definition. Furthermore, changes to software can usually be made effective only by stopping an old version of a program and starting the new one. This results in downtimes that induce high costs and possibly determine an application's success or failure. Alternatively, programming languages and runtime systems should be equipped with features that allow for subsequent unanticipated adaptations of an already active program.

Dynamic Replacement of Objects In principle, unanticipated adaptation can always be dealt with by manual redirection of references. If you want to add or replace a method in, or change the class of an object, you can simply assign its reference a new object with the desired properties. However, there are two consistency problems involved in this approach on the conceptual level. Firstly, if there is more than one reference to an object, they all must be known to the programmer. Secondly, even if all references are known, they have to be redirected to the new object one by one. This approach is likely to lead to an inconsistent state if message sends to the involved objects occur in between.

It would be straightforward if we could simply “replace” an object by another one without changing the involved references, and thereby avoid these consistency problems. In the following sections we outline the programming language GILGUL and its new operations that allow for this kind of atomic object replacement. We show that a dissection of the concept of object identity and a strict separation of the included notions of reference and comparison is needed in order to define GILGUL's semantics in a clean way. An example of use illustrates the feasibility of this approach.

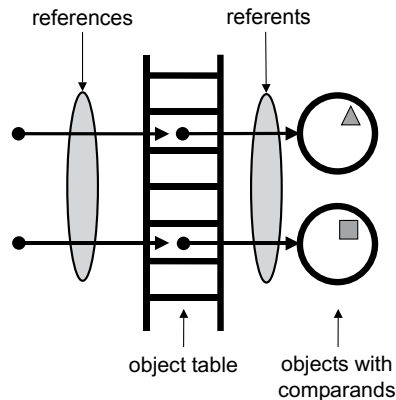


Fig 1. References to objects are realised as OOPs, and each object stores a comparand.

2 Gilgul

The Gilgul Model GILGUL’s model can be illustrated with an implementation technique called “Identity Through Indirection” in [6]: Here, a reference to an object is realised as an object-oriented pointer (OOP). An OOP points to an entry in an object table which holds the actual memory addresses. Since we do not want to restrict our model to this implementation technique, we abstract from this terminology and say that object references point to entries which hold *referents* that represent the actual objects (see figure 1).

In GILGUL, references are never compared. To be able to compare objects we combine “Identity Through Indirection” with “Identity Through Surrogates” [6]. Each object is supplemented with an attribute that stores a *comparand*. Comparands are system-generated, globally unique values that cannot be manipulated directly by a programmer. The comparison of objects ($o1 == o2$) then means the comparison of their comparands ($o1.comparand == o2.comparand$), but they are never used for referencing.¹

The Programming Language Gilgul This scheme provides the conceptual basis for the programming language GILGUL. It introduces means to manipulate referents and comparands and has been carefully designed not to compromise compatibility with existing Java sources.

Operations on Referents In GILGUL, the *referent assignment operator* $\# =$ is introduced to enable the proposed replacement of objects. The referent assignment expression $o1 \# = o2$ lets the referent of the variable $o1$ refer to the

¹In [6], this additional attribute is named *surrogate*. In our context, this term might raise the wrong associations. Therefore, we have opted for the artificial word *comparand* to stress that this attribute is a passive entity that is strictly used within comparison operations only.

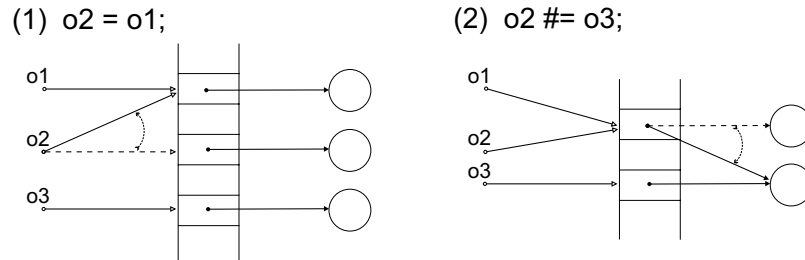


Fig 2. Referent Assignment: After execution of `o2 #= o3`, all three variables refer to the same object. Since `o1` holds the same reference as `o2`, it is also affected.

object `o2` without actually changing any references. Effectively, this means that all other variables which hold the same reference as `o1` refer to the object `o2`, too. Consider the following statement sequence.

```
o1 = new MyClass();
o2 = o1;
o2 #= o3;
```

After execution of the referent assignment, all three variables are guaranteed to refer to the same object `o3`, since after the second assignment, `o1` and `o2` hold the same reference (see figure 2).

Figure 2 also illustrates why a strict separation of reference and comparison is needed in order to allow for this kind of manipulations. Assume that you want to compare `o2` and `o3` in the scenario on the right-hand side of figure 2. Here, comparison of variables without the use of comparands is ambiguous on the conceptual level, since comparison of the references would yield `false`, whereas comparison of the referents would yield `true`. Therefore, one has to opt for comparison of properties that are stored inside of the involved objects in order to make comparison of variables unambiguous.

Operations on Comparands Comparands are introduced in GILGUL by means of a (final) pseudo-class `java.lang.Comparand` which can be used to create new comparands via class instance creation expressions (`new Comparand()`). By default, the definition of `java.lang.Object` includes an instance variable `comparand` of this type. The definition of the equality operators `==` and `!=` is changed in GILGUL, such that `o1 == o2` means the same as `o1.comparand == o2.comparand`, and `o1 != o2` means the same as `o1.comparand != o2.comparand`.

There are good reasons on the conceptual level to allow for copying of comparands between objects. For example, decorator objects may “take over” the comparand of the decorated object as follows: `o1.comparand = o2.comparand`. This ensures that comparison operations yield the correct result even when “direct” references to the decorated object are involved.²

²Other usage scenarios are given in [3].

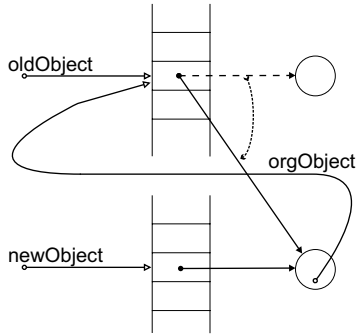


Fig 3. Naive application of `oldObject #= newObject` might lead to an unwanted cycle.

There are no limitations on the concrete implementation of comparands. For example, a reasonable implementation is a 64-bit integer with comparands being created by increment of a global counter. This scheme provides for roughly 10 billion unique comparands per second for half of a century. Still, the actual implementation is completely hidden from programmers. Especially, GILGUL prevents comparands from being cast to any other type and, for example, does not allow for their involvement in arithmetic operations.

Example of Use We are now able to achieve the desired atomic replacement of objects. We can apply `oldObject #= newObject` to let `newObject` replace `oldObject` consistently for all clients that have references to `oldObject`. However, care has to be taken when `newObject` refers to `oldObject` in order to delegate messages that it cannot handle by itself. Regard the following naive sequence of operations.

```
newObject.orgObject = oldObject;
oldObject #= newObject;
```

This would be erroneous, because afterwards `newObject.orgObject` would refer to `newObject`, since it contains the same reference as `oldObject` according to the first assignment. This, of course, leads to a cycle and therefore, to non-terminating loops for messages that cannot be handled by `newObject` (see figure 3). The following statement sequence however is correct (see figure 4).

```
tmp #= oldObject; // let a new reference refer to oldObject
newObject.orgObject = tmp; // use tmp instead of oldObject for forwarding
newObject.comparand
  = oldObject.comparand; // ensure that equality behaves well
oldObject #= newObject; // tmp and so newObject.orgObject are not changed
```

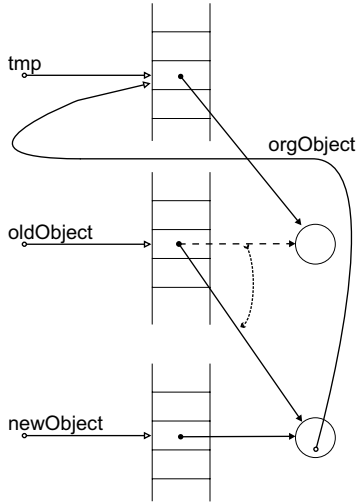


Fig 4. The use of a temporary variable avoids the unwanted cycle.

The actual “replacement” of `oldObject` is initiated by the last operation, and thus is indeed atomic. As we can see, GILGUL’s new operations give the programmer the possibility to “replace” the former object atomically and thereby relieves him/her from having to deal with any consistency problems. Furthermore, the involved objects need not anticipate such modifications, reducing the complexity of the development of actual components to a great extent.

3 Conclusions

We have designed the programming language GILGUL, a compatible extension to Java. It introduces the pseudo-class `Comparand` and the referent assignment operator `#=`. It also changes the definition of the equality operators `==` and `!=` accordingly. We have shown how GILGUL’s new operations can be applied for the purpose of atomic and consistent object replacement.

GILGUL is the first approach known to the author that strictly and cleanly separates the notions of reference and comparison on the level of a programming language.³ Note that this separation of object identity concerns is orthogonal to the usual distinction between reference semantics and value semantics that is found various programming languages, for example [1, 7, 8]. These languages allow programmers to choose from these semantics when comparing variables, but they all still rely on comparison of references in order to establish object identity. For this reason, object identity usually coincides with reference semantics. In contrast, our approach relies on comparands to determine (logical) identity, and this opens up new degrees of flexibility.

³An overview of related work centered around the theme of object identity is given in [4].

There are several issues in the design of GILGUL that have been discussed in previous publications. They include: the relation to Java's standard `hashCode()` method [4], declaration of restrictions on the applicability of GILGUL's new operations [2, 4], and the introduction of implementation-only classes to allow for subtractive replacements [2]. Another issue is the semantics of dynamic object replacement in the presence of active methods; details will be reported elsewhere.

Currently, a compiler and runtime system for GILGUL is being developed at the University of Bonn. We attempt to include optimizations, for example the use of direct pointers to an object as long as possible until an extra level of indirection is inevitable, and the subsequent reversal to direct pointers as soon as possible. Future work also includes a formal proof of type soundness.

References

- [1] K. Arnold and J. Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, 1998.
- [2] P. Costanza. *Dynamic Object Replacement and Implementation-Only Classes*. accepted for: *6th International Workshop on Component-Oriented Programming (WCOP 2001)* at ECOOP 2001, Budapest, Hungary.
- [3] P. Costanza and A. Haase. *The Comparand Pattern*. accepted for *EuroPLoP 2001*, Irsee, Germany.
- [4] P. Costanza, O. Stiemerling, and A. B. Cremers. *Object Identity and Dynamic Recomposition of Components*. in: *TOOLS Europe 2001*. Proceedings, IEEE Computer Society Press.
- [5] P. Grogono and M. Sakkinen. *Copying and Comparing: Problems and Solutions*. in: *ECOOP 2000*. Proceedings, Springer.
- [6] S. N. Khoshafian and G. P. Copeland. *Object Identity*. in: *OOPSLA '86*. Proceedings, ACM Press.
- [7] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
- [8] D. N. Smith. *Smalltalk FAQ*. <http://www.dnsmith.com/SmallFAQ/>, 1995.