WINNER84
Winner, Robert I. "Unassigned Objects". *ACM Transactions on Programming Languages and Systems,* Volume 6, Number 4 (October 1984), pp. 449-467. This paper discusses language design issues arising from considering objects that might not have a value. This is an extention of the problem of uninitialized variables in that Winner also considers objects that may lose their value during their lifetime. Objects with no current value are called "unassigned objects." These problems are discussed in the context of assignment oriented languages in general, and Ada in particular. Ada was chosen because it is supposed to be used to create reliable software, its development reflected a change in the language's handling of unassigned objects, and because it is large enough to show the interactions between design decisions on unassigned objects and other language features. The author poses various questions to be answered, and then discusses them. He considers the reasons for thinking about unassigned objects, and describes the various circumstances under which an object may not have a value. He proposes several language facilities to allow the programmer to explicitly control the "assignedness" of an object. He discusses scalar objects and composite objects separately. He shows that scalar objects are relatively easy to handle, but that composite objects are more difficult. He attributes this to the "dual view" of composite objects - they are sometimes considered as being composed of smaller objects, and sometimes considered as objects in their own right. This causes both conceptual and implementation problems. Finally, he discusses when using an unassigned object is an error, and presents his conclusion that the issue is more complicated than it looks and that it is often not given enough attention in language design. Winner considers this an issue in designing languages that support the construction of reliable software. He asserts that "it is bad for a program to use the value of an object if the programmer intends the object to be logically unvalued". He takes this as justification for including into a programming language facilities for a programmer to indicate that a variable should no longer have a value, and for defining as an error certain uses of an object that does not currently have a value. For scalar types, this is done by extending each type with a distinguished value called *unassigned.* He also proposes a new "unassignment statement" that copies the value of a variable, and then makes it *unassigned.* He also discusses the propagation of an unassigned value from one object to another. He briefly considers various implementations of "unassigned", showing that even if it is possible to represent the value at the hardware level, there still can be implementation problems. Composite objects are shown to present more difficult problems. The basic question is: when a composite object has some components with values and some *unassigned* components, is the composite object as a whole considered to have a value or to be *unassigned?* He shows that most of the problems arise from the "dual view" described above. He mentions the idea of eliminating the dual view, but appears to be reluctant to conclude that it is the best solution. Finally, he considers the use of unassigned objects - when should it be an error? He concludes that the problem of unassigned objects should be explicitly addressed in all assignment oriented language designs, even if the decision is to ignore the problem. Students of language design should read this paper. It shows how a minor issue can interact in unexpected ways with other language design decisions to become more complex than it appears at first glance.