0. Abstract.

We believe that axiomatic reasoning about program behaviour should be based directly on a semantic model specifically tailored for that purpose. Moreover, the structure of the semantic model should be used directly to suggest the structure of an assertion language for expressing program properties. It is desirable, therefore, to adopt a semantics with as clean and simple a structure as possible, so that one can use assertions with simple syntactic structure and build a clean and simple axiomatic proof system for program properties. By basing the proof system closely on the underlying semantic model, one is able to use the semantic model directly in establishing the soundness and relative completeness of the proof system; these tasks are made less difficult if the semantics has a simple structure. We illustrate these ideas by applying them to a small programming language, a simple block-structured imperative language which allows sharing or aliasing among identifiers. Although the language is rather simple and is certainly far from being a fully fledged programming language, it exhibits enough semantic features to merit a detailed semantic investigation and serves well to illustrate our methodology for designing an axiomatization. We first define a standard semantics and discuss the full abstraction problem for this language. We define a semantic relation called sharing equivalence, and show that the standard semantics is fully abstract "up to sharing equivalence". We then construct a semantics based directly on sharing equivalence classes, which is fully abstract. We establish some important semantic properties, and use them in designing an axiomatic proof system for partial correctness properties of programs.

1. Introduction.

The first part of the paper introduces the syntax and a standard denotational semantics for a simple block-structured programming language which allows sharing or aliasing. Sharing arises naturally in procedural languages which permit certain forms of parameter passing (like call-by-reference): two identifiers share or are aliases if assignment to one affects the value of the other. Here we do not include procedures in our language, but instead include a form of declaration that introduces aliasing among program identifiers explicitly. This allows us to focus on the semantic treatment of aliasing in a simpler context than a fully fledged procedural language. This has the advantage that a satisfactory treatment of aliasing is possible with a relatively straightforward semantics and axiomatic system. Nevertheless, we must admit that in a language with procedures our treatment would need to be modified extensively. Even in this restricted setting we believe that our approach is of interest.

We begin with a conventional semantic model suitable for the proper modelling of aliasing, involving locations, stores, and environments. Identifiers which are aliases are mapped by the environment to the same location, and an assignment to an identifier affects the value stored in the corresponding location. Hence the common description: identifiers which are aliases share a memory location. We define the semantics relative to an abstract storage allocation primitive (as usual called new), assumed only to possess a natural "newness" property. We discuss several variant semantics, differing only in details of storage handling.

We next state some well known and fairly obvious properties of commands, expressions, and declarations. Their semantics is determined completely by their dependence on and effect upon identifiers occurring (free) in their text. In proving these results we introduce a natural notion of sharing equivalence on environment-store pairs, which plays a crucial role in the rest of the paper. In order later to justify the soundness of an inference rule involving change of bound variables in blocks, We introduce syntactic substitutions as renamings of identifiers in program terms. With the appropriate notions of substitution of free identifiers