We give a new denotational semantics for a shared-variable parallel programming language and prove full abstraction: the semantics gives identical meanings to commands if and only if they induce the same behavior in all program contexts. The meaning of a command is a set of "transition traces," which record the ways in which a command may interact with and be affected by its environment. We show how to modify the semantics to incorporate new program constructs, to allow for different levels of granularity or atomicity, and to model fair infinite computation, in each case achieving full abstraction with respect to an appropriate notion of program behavior. © 1996 Academic Press, Inc.

1. INTRODUCTION

One of the fundamental purposes of semantics is to provide rigorous means of proving the correctness of programs with respect to behavioral specifications. For any particular language different semantic models may be suitable for reasoning about different behavioral notions, such as partial correctness, total correctness, deadlock-freedom, and more general liveliness and safety properties [14]. Ideally one would like a semantics in which the meaning of one term coincides with the meaning of another term if and only if the terms induce the same behavior in each program context; this guarantees that one term may be replaced by the other in any context without affecting the behavior of the overall program, thus supporting compositional or modular reasoning about program behavior. Such a semantics is said to be equationally fully abstract with respect to the given notion of behavior [17, 20, 22]. When the set of program behaviors is equipped with an approximation ordering and the semantic model has a partial order such that the meaning of one term is less than the meaning of another if and only if the behavior of the first term in each program context approximates the behavior of the second term in the same context, the semantics is said to be inequationally fully abstract with respect to the given notion of program behavior and approximation. Clearly an inequationally fully abstract semantics is also equationally fully abstract. Intuitively, a fully abstract semantics is at exactly the right level of abstraction to support compositional reasoning.

The difficulty of finding fully abstract semantics is well known [4, 17, 20, 22]. Many standard semantic models are correct, in that whenever two terms induce different behavior in some context they denote different meanings, but too concrete since the converse may fail. Sometimes one can show that by adding extra syntactic constructs to the programming language the model becomes fully abstract. However, unless the extra constructs are computationally natural and the original language was clearly deficient because of their omission, the full abstraction problem for the original language is still important.

The standard state-transformation semantics for sequential while-programs is fully abstract with respect to partial correctness behavior. However, for a parallel version of this language [9, 18], in which parallel commands can interact by updating and reading shared variables, the full abstraction problem is more difficult. Parallel programs may exhibit non-deterministic behavior, depending on the scheduling of atomic actions, so the partial correctness behavior of a parallel command is naturally modelled as a non-deterministic state transformation, usually represented as a function from states to sets of states. However, the state transformation denoted by a parallel combination of commands cannot be determined solely from the state transformations denoted by the component commands; thus the state-transformation semantics for a parallel language is not even compositional, and is certainly not fully abstract. It is not even sufficient to model a program as a set of sequences of states, each sequence recording a possible execution history of the program, since this semantics still fails to be compositional. One needs a semantic model with more detailed structure, so that the possible interactions between commands executing in parallel may be modelled appropriately.

Hennessy and Plotkin [9] described a denotational semantics for this language, based on a recursively defined domain of resumptions, built with a powerdomain operator. However, the resumptions semantics is too concrete: skip and skip; skip denote different resumptions even though they induce the same partial correctness behavior in all contexts. They showed that with the addition of extra features to the programming language, the resumptions model becomes fully abstract. However, one of the extra constructs is a rather peculiar form of coroutine execution which allows counting of the number of atomic steps taken by a