A CONSTRUCTIVE APPROACH TO COMPILER CORRECTNESS *

Peter Mosses
Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Aarhus C, Denmark

Abstract
It is suggested that denotational semantic definitions of programming languages should be based on a small number of abstract data types, each embodying a fundamental concept of computation. Once these fundamental abstract data types have been implemented in a particular target language (e.g. stack-machine code), it is a simple matter to construct a correct compiler for any source language from its denotational semantic definition. The approach is illustrated by constructing a compiler similar to the one which was proved correct by Thatcher, Wagner & Wright (1979). Some familiarity with many-sorted algebras is presumed.

1. INTRODUCTION

There have been several attacks on the compiler-correctness problem: by McCarthy & Painter (1967), Burstall & Landin (1969), F.L. Morris (1973) and, more recently, by Thatcher, Wagner & Wright, of the ADJ group (1979). The essence of the approach advocated in those papers can be summarised as follows: One is given a source language L, a target language T, and their respective semantics in the form of models M and U. Given also a compiler to be proved correct, one constructs an encoder: M → U and shows that this diagram commutes:

\[
\begin{array}{ccc}
L & \overset{\text{compile}}{\longrightarrow} & T \\
\downarrow & & \downarrow \\
M & \overset{\text{encode}}{\longrightarrow} & U \\
\end{array}
\]

*) An earlier version of this paper is to appear in Proceedings of a Workshop on Semantics-Directed Compiler Generation, Aarhus, 1980 (Springer-Verlag).
It is assumed that the semantic and compiling functions are "syntax-directed". This amounts to insisting on denotational semantics in the style of Scott & Strachey (1971): "The values of expressions are determined in such a way that the value of a whole expression depends functionally on the values of its parts". ADJ (1979) reformulated this in the framework of initial algebra semantics, where the grammar, say G, of L is identified with "the" initial G-algebra. The advantage of this is that a semantic function: L \rightarrow M can be seen to be a (by initiality, unique) homomorphism from L to a G-algebra based on the model M. Similarly, a compiling function: L \rightarrow T is a homomorphism from L to a G-algebra derived from T, and then the semantics: T \rightarrow U induces a G-algebra based on U.

So L, M, T and U can be considered as G-algebras, and the two semantics and the compiler are homomorphisms. A proof that encode: M \rightarrow U is a homomorphism then gives the commutativity of the above diagram, by the initiality of L. (Actually, to interpret this as "compiler correctness", one should also show that encode is injective, or else work with decode: U \rightarrow M.) ADJ (1979) illustrated the approach for a simple language L, including assignment, loops, expressions with side-effects and simple declarations. T was a language corresponding to flow charts with instructions for assignment and stacking. Their semantic definitions of L and T can be regarded as "standard" denotational semantics in the spirit (though not the notation!) of Scott & Strachey (1971). They succeeded in giving a (very!) full proof of the correctness of a simple compiler: L \rightarrow T.

We shall take a somewhat different approach in this paper. The semantics of the source language L will be given in terms of an abstract data type S, rather than a particular model. The target language T will also be taken as an abstract data type. Then the correct implementation of S by T will enable us to construct a correct compiler (from L to T) from the semantic definition of L. The compiler to be constructed is actually the composition of the semantics and the implementation, as shown by the following diagram:

The models M and U are not relevant to the proof of the correctness of the implementation: S \rightarrow T, but may aid the comparison of this diagram with the preceding one.