ABSTRACT

The correctness of programs with programmer-declared functions is investigated. We use the framework of the typed lambda calculus with explicit declaration of (possibly recursive) functions. Its expressions occur in the statements of a simple language with assignment, composition and conditionals. A denotational and an operational semantics for this language are provided, and their equivalence is proved. Next, a proof system for partial correctness is presented, and its soundness is shown. Completeness is then established for the case that only call-by-value is allowed. Allowing call-by-name as well, completeness is shown only for the case that the type structure is restricted, and at the cost of extending the language of the proof system. The completeness problem for the general case remains open. In the technical considerations, an important role is played by a reduction system which essentially allows us to reduce expression evaluation to systematic execution of auxiliary assignments. Termination of this reduction system is shown using Tait's computability technique. Complete proofs will appear in the full version of the paper.
We present a study of partial correctness of programs with programmer-declared functions. Typically, if "fac" is declared as the factorial function, we want to be able to derive formulae such as \( \{ x=3 \} y := \text{fac}(x) \{ y=6 \} \). For this purpose, we use a functional language with an interesting structure, viz. the typed lambda calculus together with explicit declaration of (possibly recursive) functions - rather than using the fixed point combinator - and then consider a simple imperative language the expressions of which are taken from this functional language. The reader who is not familiar with the typed lambda calculus may think of function procedures as appearing in ALGOL 68, provided only finite (not recursively declared) modes are used.

Section 2 first introduces the syntax of our language(s). As to the functional language, besides constants and variables it contains application, two forms of abstraction, viz. with call-by-value and call-by-name parameters, and conditional expressions. The imperative language has assignment, composition and conditional statements. A program consists of a statement accompanied by a list of function declarations. The assignment statement constitutes our main tool in applying a formalism in the style of Hoare to an analysis of correctness of programs with function procedures. A central theme of the paper is the reduction of expression evaluation to execution of a sequence of assignment statements, thus allowing the application of the well-known partial correctness formalism for imperative languages. Some further features of our language are: function evaluation has no side-effects, the bodies of function declarations may contain global variables, and the static scope rule is applied. Section 2 also provides a denotational semantics for the language, with a few variations on the usual roles of environments and states, and applying the familiar least fixed point technique to deal with recursion.

Section 3 presents an important technical idea. A system of simplification rules is given for the statements of our language allowing the reduction of each statement to an equivalent simple one. These rules embody the above-mentioned imperative treatment of expression evaluation, and play a crucial role both in the definition of the operational semantics to be given in Section 4, and in the proof systems to be studied in Sections 5 to 7. The proof that the reduction always terminates is non-trivial. Details are given in the Appendix; the proof relies on the introduction of a norm for each expression. The existence of this norm is proved using an auxiliary reduction system. Reduction in this auxiliary system always terminates as is shown using the "computability" technique of Tait [22]. In Section 4 we define an operational semantics for our language and prove its equivalence with the denotational one.

In Section 5 the notion of partial correctness formula is introduced, and a sound proof system for partial correctness is proposed. The techniques used in the soundness proof rely partly on the equivalence result of Section 4, partly follow the lines of De Bakker [4]. In Section 6 we show that a slight modification of the proof system is complete for a language with only call-by-value abstraction. This is shown