Algebraic Modelling of Imperative Languages with Pointers

A. V. Zamulin

Institute of Informatics Systems
Siberian Division of the Russian Academy of Sciences
630090, Novosibirsk, Russia
e-mail: zam@isi.itfs.nsk.su

Summary. A modelling technique for an imperative programming language is presented in the paper. The technique is based on the algebraic approach to the data type specification and introduces such basic data types as memory, variable, record, function, and procedure and such basic kinds of expressions as linear and conditional expressions. It is shown that statements are a partial case of expressions, pointers are a partial case of variables, and procedures are a partial case of functions. The technique is illustrated by a simple Pascal program including variables, pointers, and assignment and conditional statements.

1 Introduction

All the problems of program verification, transformation, partial evaluation, and abstract interpretation require a formal representation of a program as a mathematical entity. The task is easier with respect to functional languages based on the conventional mathematical notions. It is not so in the case of imperative languages. The reason is that the computation model of such a language is based on a notion of memory. It means that both input data and intermediate data of a program are stored in memory for some time and used when needed. Therefore, we need to formally specify the memory and associated objects and tie these notions in with other notions (statement, function, procedure, etc.) of an imperative programming language.

One of the most promising approaches to the definition of programming languages is an algebraic one which is studied for a quite a long time [Let68, GTY74, GTW78, BWP87]. It was successfully applied to functional languages [BW82]. The attempts taken in the algebraic modelling of imperative programming languages are, to our mind, far from success. It is simply proposed that "programming languages should be studied in terms of algebraic theories" in [GTW78], for example. A many-sorted algebra is used in [Let68] to construct a general theory of a formal language. A special case of the many-sorted algebra, a multialgebra, is used in [GTY74] for the formal definition of several specific data structures.

* Author's present address: School of Computer Science, University Sains Malaysia, 11800 Penang, Malaysia; e-mail: zam@cs.usm.my
A substantial effort to algebraically define an imperative programming language was made in [BWP87] where it is regarded as a hierarchy of abstract data types (ADT) constituting a theory of many-sorted algebras. Such data types, as "expression", "statement", "loop", etc., are introduced in the paper to represent the most popular syntactic constructs of an imperative language. The paper does not give, however, a specification of memory and related concepts.

An advanced work in this field is represented by [Wag84]. It gives the definitions of the most important data types of an imperative programming language in terms of category theory. However, this author, reasonably criticising "the data theorists" for introducing models which are very far from the usual programmers' reality, offers such tricky things which are also far away from the usual programming. The model becomes even more complex when pointers are introduced [Wag87].

A simple model disregarding pointers and locations was proposed in [Don77]. Unnamed locations are necessary however when complex objects with computed addresses of their components are considered. Such a computed address is often used as the left hand side of an assignment statement or procedure argument called by reference. Therefore, a model of an imperative programming language is needed which, being abstract as much as possible, could simulate the conventional language notions in terms close to the practice of a usual programmer. One of the main tasks of the model is incorporation of pointers and unnamed locations so that they become a natural constituent of a programming language. An attempt to define pointers by means of the algebra of partial maps is undertaken in [Moe93]. Another version of pointer definition is given in [BBB85] where they are associated with a special structure "plexus". Our approach differs from these two by considering pointers as a special case of variables, thus avoiding the necessity of using an extra algebra or an extra data structure.

The classical algebraic theory is always a triple "set of sort names, set of sort-indexed operators, set of axioms". In programming languages, this triple is usually divided into several groups each consisting of a sort name and corresponding operators and axioms. Such a group is called an abstract data type (ADT). The theory is regarded then as a family of abstract data types. This approach is followed in this paper which is organised in the following way.

The definition conventions are discussed in Section 2. The major concepts of the paper, the memory and variable data types, are specified in Section 3; they are illustrated by a sample Pascal program. Several forms of expressions are introduced in Section 4. A record type is specified in Section 5 and a function type is specified in Section 6. Some conclusions are drawn in Section 7.