Constructing a Theory of a Data Structure as an Aid to Program Development

C. B. Jones

IBM European Systems Research Institute, International Education Centre,
Chaussée de Bruxelles 135, B-1310 La Hulpe, Belgium

Summary. This paper illustrates an extension to the method of developing programs via abstract data types. In order to make the proofs shorter and more intuitive a collection of lemmas (theory) is constructed for the main data types (trees). The problem used as an example is the recording of equivalence relations, one of the programs given is based on the Fischer-Galler algorithm.

Introduction

The aim of any systematic approach to program development can be viewed as increasing our confidence in the correctness of the programs which we develop. This paper attempts to show the contribution that a particular way of using 'data abstraction' can make towards this goal.

The idea of using abstract data types (e.g. sets or, as used below, collections of trees) in program specification and design is now widely recognized as contributing to clear and correct development. However, the correctness proofs for development steps can become long and detailed. The proposal exhibited in this paper is the development of a theory of the abstract data type(s) used. This has two advantages. Firstly, subsequent proofs become both shorter and closer to our intuitive level of discussing programs. Secondly, the results which have been collected about the data structure itself will be available for use in other problems employing similar objects. These two points could result in a dramatic increase in the size of problem to which proof techniques can be applied.

Let us consider our starting point more carefully. (For readers to whom this method is unfamiliar, the example below will provide examples of the steps involved.) The idea of using data abstraction as a way of designing programs was shown in early, informal, papers of E.W. Dijkstra. Formal treatments are given in [17, 9, 10] and [11]. The basic idea is applicable both to specifying a program or system and to the development of programs which satisfy such specifications. In defining some required function, it is proposed that the
specification should use objects which match the problem. While it is difficult to propose a formal test for such 'matching', it should be clear that the use of objects like sets or trees is likely to facilitate much more succinct specifications than would be possible if they were described in terms of representations of these 'abstract objects' at a machine level (e.g. in terms of bits and bytes). A specification using abstract objects will be much easier to reason about and its intention should be more readily related to the given problem.

There are two basic approaches to specifications using abstract objects: implicit and constructive. Much of the work on implicit specifications is being done in an algebraic framework (cf. [7, 15, 6]). This author has explained elsewhere [14] how a test of constructive specifications can be used to eliminate any 'bias' towards particular representations that they might have. This dichotomy is not pursued here because the approaches to proving representations correct are similar (cf. use of 'QREP' in [8]).

A constructive specification will use combinations of known data types (e.g. sets) and, possibly, limit the combinations by a 'data structure invariant'. A representation will again be built from data types; but, rather than being chosen for their abstractness, they will be selected because they are closer to the machine or language on which the required program is to run. In general, the representation will have more detail than the abstraction (e.g. lists have ordering information absent from sets) and the natural way to express their relationship is by a function from the representation to the abstraction. Hoare [9] calls these 'abstraction' functions, here we follow [12] and refer to them as 'retrieve' functions.

The first requirement for a representation is that it be 'adequate' to represent all possible values of the abstraction. That is, taking into account their data structure invariants, for each element in the abstraction there must be at least one element in the representation which the retrieve function maps into the former element. Secondly, it is necessary to show that the proposed operations on the representation 'model' those on the abstraction. For simple functions this can be shown by commutativity with the retrieve function. The situation is slightly more complicated where the operations – either on the abstraction or the representation – are specified by post-conditions permitting a range of results. As has been implied the development may be made in several stages. The solution (i.e. representation) at one stage becoming the specification (i.e. abstraction) at the next.

Having described the state-of-the-art, we can now explain the refinement of this method which is exemplified in the current paper. In proving results about a specification or about the relation of an implementation thereto, properties of the data structures are required. If the abstract data types are all from known algebras (e.g. sets) the required properties may well be known within their algebras. On the other hand a specification written, for example, in terms of trees may be a difficult base precisely because the algebra is not well-known. Proofs which are undertaken without an established set of results are likely to become long and opaque because the specific results are being developed along with the general theory of the objects. The technique which is to be exemplified below will show the advantage of proving results about the