Homomorphisms and Promotability

Grant Malcolm

Department of Computing Science
Groningen University
P.O. Box 800
9700 AV Groningen
The Netherlands

Abstract

The construction of structure-preserving maps, or "homomorphisms," is described for an arbitrary data type: examples of these functions are given for list- and tree-like structures and types defined by mutual induction. From the definition of a data type it is also possible to infer a "promotion" theorem for proving equalities of homomorphisms.

1 Introduction

Our starting point in this paper is the "Bird-Meertens Formalism" currently being developed by Richard Bird at Oxford and Lambert Meertens at Amsterdam. This formalism comprises a concise functional notation and a very few, powerful theorems for proving equalities of functions. The conciseness of the notation is largely a result of concentrating attention upon two basic operations and — until recently — upon only one data structure: finite lists. Using the basic operations, it is possible to construct a large number of list manipulating functions; the basic nature of the operations ensures that the resultant functions are potentially both easily understood and grossly inefficient. Both comprehensibility and inefficiency can then be reduced in a process of program transformation which results in an equivalent and, ideally, more concise and efficient program.

It is in this process of program transformation that the advantage of concentrating upon only a few theorems for proving equalities becomes apparent. The ideal is to have only one theorem applicable at any given stage in the transformation, and to have that theorem's applicability be obvious. Such an ideal is unlikely to be realised, but the general principle is: the more trivial and mechanical the transformation process can be made, the better. For then the construction and implementation of tactics to handle the trivial steps is made easier, leaving one free to devote one's energies to those stages which require some degree of ingenuity.

Recent work by Bird and Meertens [3,8] has applied the formalism to other data structures such as binary trees, arrays and "rose trees." It turns out that the basic operations and theorems used in the work on finite lists have analogous operations and theorems which are applicable to those other data structures. So, rather than building "a theory of lists," "a theory of rose trees," and so on, might it not be possible to build a polymorphic theory of data structures, a theory applicable to any given data structure? The answer we give in this paper, albeit hesitantly, is "yes." We concentrate upon only one operation, which is polymorphic in that there is an analogous operation for every data structure: these operations are called "homomorphisms" by Bird (and "catamorphisms" by Meertens — we shall adopt the former term). Furthermore, we concentrate upon only one theorem for proving equalities of functions, which is a polymorphic theorem in the sense that there is a form of this theorem for every data structure.

Δίγα δέντρα και λίγα
Βρεμένα χαλίκια
— Odysseus Elytis, Επέτειος.
and this form may be inferred in a uniform way. These theorems are called "promotion theorems" by Bird and Meertens; we shall adopt this term as well, although our theorems are more general.

The implication that there does not already exist a polymorphic theory of data structures might cause offence: such a theory is certainly implicit in Bird's and Meertens' recent work; the definition we give of homomorphism is similar to definitions of terms in the second-order polymorphic \( \lambda \)-calculus [5]; and should a category theorist happen to read this paper, his response would surely be "well yes, we knew that already."

There is a close connection between the results of this paper and certain basic notions in category theory. We shall have more to say on this connection in the concluding section, where we allow ourselves the luxury of speculation; for the present we simply note that we find the triviality of these basic category theoretical notions encouraging — as computing scientists we have already declared our interest in the trivial. The difficulty with respect to category theory lies in discovering which aspects of the theory are relevant to our own discipline. Recent work in this field has done much to clarify matters (see, for example, [6]), but there is still a long way to go.

As to the second-order polymorphic \( \lambda \)-calculus, while it was certainly a source of inspiration, it has the great drawback that one has to reason about its data structures indirectly, via a model. More general theories may be able to overcome this disadvantage, but more work has to be done in that area. In any case, we are not presenting here a formal theory of typed functions; rather, we hope to adumbrate what is common to all data structures. Constructive type theory presents a uniform mechanism for reasoning about data structures (see Backhouse [2]), and we should like to discover more such mechanisms.

Finally, we make no apology for making explicit that which previously was only implicit. The principles informing Bird's and Meertens' recent work are important enough to bear public expression, and it is our aim in the pages below to convince the reader of this.

1.1 notation

The notation we use is based on that of Bird and of Backhouse (see [4,1]). We denote the application of function \( f \) to argument \( a \) with an infix dot: \( f.a \). For binary operators we use infix notation, writing \( x \otimes y \) for the application of operator \( \otimes \) to the pair of arguments \( x \) and \( y \). If \( \otimes \) is a binary operator, then we use \( \otimes^{-1} \) to denote its "reverse," i.e.,

\[
x \otimes^{-1} y = y \otimes x.
\]

Also, we write \((x \otimes)\) for the function such that:

\[
(x \otimes).y = x \otimes y.
\]

The identity of operator \( \otimes \) (if it exists) is written: \( 1_{\otimes} \). Thus, for all \( x \),

\[
1_{\otimes} \otimes x = x = x \otimes 1_{\otimes}.
\]

We often include type information where we feel that it helps clarify an expression; if a function \( f \) takes arguments of type \( A \) into expressions of type \( B \), we denote this by \( f : A \rightarrow B \). Cartesian product of two types \( A \) and \( B \) is denoted by \( A \times B \). (Binary operators are thus considered to have type \( A \times B \rightarrow C \) for some types \( A, B \) and \( C \).) We use \( I \) for the polymorphic identity function, and let the reader infer its type from the context. We occasionally write \( a = b \in A \) to denote that \( a \) and \( b \) are both objects of type \( A \), and are equal under the equality of that type.

2 Homomorphisms and Join Lists

Central to this paper is the notion of homomorphism. The word, "homomorphism" comes from Bird (see, for example, [4]), who used it to describe a structure-preserving map from lists (an example of a monoid) to another monoid — hence the nomenclature. We use it in the more general sense of a structure-preserving map from one data structure to another, similar structure. Before giving a formal definition of homomorphisms, we exemplify the notion by considering homomorphisms over the type of join lists.