Program Transformation in Functional Programs

Masayoshi Saga

School of Information Science,
Japan Advanced Institute of Science and Technology

February 13, 1998

Keywords: program transformation, functional program, deforestation.

In functional programming, each program is composed of small and basic functions. These module-oriented programs are easy for us to read and understand. However this function composition makes many intermediate data structures such as tree and list, because those intermediate data structures are used to pass and receive information between functions. This intermediate data structure does not directly appear as a part of the result of the whole program and causes loss of efficiency.

In order to eliminate intermediate data structures, deforestation was proposed by Wadler(1990). In his deforestation procedure, 7 rules are applied to a given program repetitively and the program is transformed into efficient one. It is a simple procedure, but it requires complicated memorization to guarantee termination of the procedure.

Gill, etc(1993) capture the structure of program by $foldr$, a list operating function. They apply transformations only to programs written in terms of the $foldr$ function. Their technique does not require memorization for termination.

Deforestation without memorization, like $foldr$ technique, has advantage in a practical use but their technique is limited to lists and programs have to be abstracted by $foldr$. In addition, the well-known efficient function for reversal of list can not be transformed.

In this study, we propose partial evaluative deforestation, which does not require complicated memorization and program abstraction by $foldr$. Partial evaluative deforestation works on a call-by-value language. Furthermore, a function like the efficient list reversal function can be transformed by this technique.

The procedure of partial evaluative deforestation has two phases, classification of functions and rewriting a program to a deforested program with a newly defined function.

Functions are classified to three groups, successive function (Su), packed function (Pa) and others. A function in former two function groups is a recursive function and absolutely produces a part of the result of the whole program whenever it is called. The difference between Su and Pa is as follows.

- Su ··· All the elements of input data are equally operated.
- Pa ··· All the elements of input data are equally operated and a part of the result of the whole program is accumulated in an extra argument.

Partial evaluative deforestation mainly applies transformation to these two kinds of functions. 7 transformation rules are made by the structure of a term. Especially, a function is newly defined in four rules. These four rules follow all the patterns of function composition of Su and Pa. Basic strategies for deriving a new function are as follows.

- Unfold an inner function and apply a outer function to each right side of branch.
- Reduce if possible.
- Rewrite a function composition matching an input to a new term using a newly defined function.

In addition to these basic strategies, terms are recursively transformed by extracting an operation for the elements of data and adopting the extracted operation as a part of a newly defined function or as a part of a new term.

Equivalence of values of terms between before and after transformation and termination of the procedure are proved by structural induction on terms. Implementation of this method shows us raising an efficiency of a program because of decrease in the number of referring to data.

In spite of efficiency, termination and equivalence, partial evaluative deforestation has three major limitations. One is that types of input and output data for two kinds of function groups have to be the same. The second problem is that two kinds of functions absolutely produce a part of the result of the whole program whenever they are called. The last limitation is all the elements of input data are equally operated.

In order to extend partial evaluative deforestation, there is a serious problem. Extension of our method causes difficulty of extracting an operation for the elements of data. One of the solutions of this problem is introduction of another classification mechanism and new transformation rules based on its classified function composition. Another classification mechanism classifies functions by whether functions absolutely produce a part of the result of the whole program every time called and whether all the elements of input data are equally operated. This classification mechanism is considered to be so complicated with original classification.

As described above, partial evaluative deforestation without memorization and program abstracting by $foldr$ is proposed. Partial evaluative deforestation is for call-by-value languages. Furthermore, its effectiveness, problems and solutions are discussed. As a future work, studies on characters of functions, higer-order functions and reduction strategies remain to be done.