AIM

Association for Information and Image Management

1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910
301/587-8202

Centimeter

Inches

MANUFACTURED TO AIM STANDARDS
BY APPLIED IMAGE, INC.

1 of 1

An Informal Introduction to Program Transformation*

by

Kevin W. Hopkins

Southwest Baptist University

Bolivar, MO 65613

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

*Work performed at Argonne National Laboratory, a contract Laboratory of the
United States Department of Energy, while a Faculty Research Participant.
Program administered by the Argonne Division of Educational Programs with
funding provided by the U. S. Department of Energy.

MASTER

The submitted manuscript has been authored by a contractor of the U. S. Government under contract No. W-31-109-ENG-38. Accordingly, the U. S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U. S. Government purposes.

1. Introduction. In the summer of 1992, I had the opportunity to participate in a Faculty Research Program at Argonne National Laboratory. During that time I came in contact with some developments in computer science that were unfamiliar to me as a mathematician. Since most mathematicians have some exposure to computers, but certainly are not computer scientists, I felt it was appropriate to write a paper summarizing my introduction to program transformation. This paper is not meant to be a full explanation of the topic, but an informal introduction for the "mathematical layman."

2. A Brief History of Transformation. The idea of program transformation has existed for at least twenty years, and many different transformation systems have been developed [1, 2, 3, 5, 8, 9, 11, 16]. At Argonne National Laboratory, Dr. Jim Boyle has developed the TAMPR (Transformation-Assisted Multiple Program Realization) system and used it on a number of projects. He has also supervised a number of visitors ranging from undergraduate researchers [13] to faculty researchers.

Program transformation is a process by which a computer takes an input program and, by applying a series of correctness-preserving transformations, produces a program which does the same thing as the original program, but has been optimized. These transformations are performed in a number of small steps. Thus it is easier to prove that each step is correct and to guide the process to the desired conclusion, a correct program. Proving the correctness of the result is associated with denotational semantics [15] and is a topic that is being studied by Victor Winter, a guest of Dr. Boyle for a week this summer.

There are a number of possible definitions of the word optimized as used in the preceding paragraph. It could be that certain information was known about the

data coming to the program and the program could be made more efficient (in time, storage, or both) if this information was incorporated into the program.

Optimization could mean the program is now set up to implement a new data type such as double-precision complex or quaternion [13] that is nonexistent on most machine architectures. Optimization could mean the program is now set up to run on a different architecture from the original program. It could also mean the program is now in a different language from the original program. Dr. Boyle (and various visitors) has examined all of these types of optimization over the years. Often it is found that one optimization builds directly on the results of a previous optimization by reusing or recycling previously written transformations.

The main benefit of transformation is a reduction of programming work and an increase in correctness. Let us assume a programmer has written a complicated program or entire programming package and tested it to the point that he is confident of its correctness. Now he receives access to a much faster machine or has someone interested in using his package on a different machine architecture. Changing the program by hand to work on the new environment, and then debugging, may require almost the same amount of effort as the original writing.

Transformations provide an alternative. A set of correctness-preserving transformations could be written to perform the conversion automatically. Depending on the nature of the package, the work involved in producing the correctness-preserving transformations may be less than the work of changing the package by hand. In any case, the transformations will be reusable in any similar conversions in the future. This is in contrast to the approach of repeating all the work by hand every time a program is to be altered. A case where transformations have been used in this manner is the LINPACK software package, where fully

three-quarters of the programs were computed from others by transformations [7].

Another advantage of transformations is that they allow for the possibility to experiment with many different forms of the same program to see which is optimal [4]. Such experimentation is practically impossible when all the different versions must be produced by hand.

3. My Work at Argonne. My project at Argonne the summer of 1992 under Dr. Jim Boyle involved optimizations of the last two types: changing code to run on a different architecture and in a different language. I worked on transforming code written in pure functional Lisp (the best code to start with because it has not received any initial implementation restrictions) to Fortran code to run on distributed-memory parallel processors such as the 8-processor Intel iPSC machine (located at Argonne) and the 512-processor Intel Touchstone Delta machine (located at Caltech). To perform this project, I had to learn three things: the transformation system, the Lisp programming language, and the basics of distributed-memory parallel machines [14].

4. The Transformation System. Learning the transformation system was the easiest of these tasks. I had been at Argonne in 1982 for fifteen weeks as an undergraduate researcher and in 1983 for eight weeks following graduation to continue some work before I started graduate school in the fall. Some things had changed, but the basic ideas behind transformation and the basic procedures had not (see [6, 13] for a more extensive description of the TAMPR system). TAMPR mainly involves a pattern replacement process. The writer of the transformation specifies which pattern needs to be matched and what will replace that matched pattern.

Most of the transformations to transform the Lisp program to parallel Fortran

for distributed-memory machines had been written by Dr. Boyle and previous visiting researchers. However, the the transformations had not been fully tested. As with any programming work, the author attempts to write correct code from the start, but realizes that much testing will need to be done to ensure that the resulting code is correct. For such testing, I applied program transformations to various programs and made modifications where necessary in an attempt to ensure correctness.

4. *Lisp. The Little Lisper* [12] proved to be an excellent reference for me as I learned Lisp. (When Lisp is used in the paper, pure applicative Lisp is meant. Other versions of Lisp, such as Common Lisp, have additional properties not used in pure applicative Lisp). Lisp is a language based on Church's lambda calculus [10] and is very mathematical in its structure. However, it is much different from the programming languages with which I was familiar, such as Fortran, BASIC, COBOL, or even a programmable calculator. Perhaps the best way to show the difference is to find something fairly simple, like the n th Fibonacci number, in Fortran and Lisp. In Fortran, this can be done using a do loop, with the result being in IFIB (and IFIBM1) at the end of the loop.

```
IFIBM1 = 1
IFIBM2 = 1
DO 10 I = 2, N
    IFIB = IFIBM1 + IFIBM2
    IFIBM2 = IFIBM1
    IFIBM1 = IFIB
10 CONTINUE
```

This has a very definite assignment and a very definite looping structure. Pure functional Lisp, on the other hand, has no assignment, and the looping is done by recursion. In Lisp, the code for finding the nth Fibonacci number is

```
(defun fib (lambda (n)
  (cond
    ((eq n 0) 1)
    ((eq n 1) 1)
    (t (plus (fib (difference n 1)) (fib (difference n 2))))
  )
)).
```

The main evaluation structure in Lisp is the definition of a function, which we have done here. One specifies the arguments (here *n* is the only one) and then gives the definition. If *n* is 0 or 1, then the value 1 is returned. Otherwise, what is returned is the sum of fib of *n*-1 and fib of *n*-2, which will require more steps into the recursion to find.

There are many advantages of the using pure functional Lisp. Because Lisp has very few language constructs, the transformational procedure is greatly facilitated. According to Boyle, "Writing a pure Lisp specification ... is certainly easier, more natural, and more amenable to verification than is writing the Fortran version directly" [5, p. 407]. The functional capability of Lisp ensures that each call to the same function with the same arguments will produce the same result (as a mathematical function should). Thus the result of one function evaluation will affect another function evaluation only if the first function result is an argument of the second function. It is not unusual for a Lisp function to have multiple Lisp functions as its arguments. Therein lies the parallelism [14] that is exploited in the

conversion to Fortran. Each argument that is a function evaluation can be evaluated independently of the others. This is what parallel evaluation means: two or more tasks can be done independently of each other and hence done in parallel. This parallelism is obvious in a Lisp program, but is often very difficult to find in a standard Fortran program (see [5] for the resulting parallel Fortran code as compared to the Lisp code for the Fibonacci example above).

The main data structures in Lisp are atoms and lists (the actual data structure is not needed in writing the specification, but is needed when considering storage requirements and how the program is transformed to Fortran). Lists are much like a set and have unspecified length (as opposed to Fortran, where the length of an array must be prespecified). This fact makes the storage representation in Lisp much different from that in Fortran. The memory of the computer is segmented into regions, each region holding certain types of data. See Figure 1 for a diagram of this memory for the case of a single processor.

Each list is stored by storing the addresses or pointers of the two parts: the car, or first part of the list and the cdr, or rest of the list. For many operations all that is needed is information about the car or cdr (such as, is it an atom, a list or nil?). This can be determined from the examining only the address due to the regions of the memory. Each atom is stored in only one spot, even if it appears in multiple lists. Arithmetic operations with integers have to take account for the fact that zero is stored at the address numadd, so one cannot just add the addresses of two integers to find their sum.

It is the storage of lists and atoms that makes the distributed-memory parallel version more difficult than the global-memory parallel version (where the storage of lists and atoms is just like the single-processor version). To access data, one

must know not only the address, but the processor, or node, that has the data. Thus the data must be split into two pieces, one piece indicating the node and the other the address on that node.

If one processor needs data that another one has, a message must be sent to request that data, and the first processor must wait for the return of the data (or suspend that operation and work on another task until the data is available). However, to ensure that the same atom is stored only once, all atoms must be stored on one processor. The data representation that was finally agreed upon is given in Figure 2. Integers (and real numbers when implemented) will have an address of themselves and will be stored with a nodeid of `intnode` (or `realnode`) to distinguish them from lists with the same address. With this representation, the three basic questions (do we have an atom, list, or nil?) can still be determined by examining the address and/or the nodeid.

5. Results. My main task was to test and improve the transformations (written mainly by Jim Boyle; David Paik, a student at Argonne in summer 1991; Jeanine Gild, a student at Argonne in spring 1992; and Peter Schuetz, a foreign visitor to Argonne in 1988) to transform pure Lisp to Fortran for distributed-memory parallel machines. As with any programming task, debugging is often more difficult than producing the original code. The debugging task was made somewhat more difficult when the decision was made to convert the existing transformations to a new grammar during the summer. Thus there were two possible sources of errors: errors in the original transformation and errors in the conversion to the new grammar.

To prepare for distributed-memory machines, all data was split into two parts: one part for the nodeid of the data and the other part for the address of the data. I

tested this split derivation on sequential machines to be sure it was working properly before the parallelism issues were added. Time did not allow me to test the programs on any parallel machines, although I did spend time testing the transformations to produce the code.

6. Conclusions. Transformation systems are beginning to pay dividends after years of development. With new computer technology developing almost more rapidly than it can be used, maintaining a library of programs that is both accurate and efficient for the new technology is a formidable challenge. Program transformation techniques can prove very useful to this end. The future of this area is bright. I think we will all be amazed at the advances that will come about in the near future.

Figure 1

Single Memory Lisp Storage

Figure 2

Multi-Processor Memory Lisp Storage

References

1. J. J. Arsac, "Syntactic source to source transformation and program manipulation," *Communications of the ACM* 22(1), pp. 43-54 (Jan. 1979).
2. R. Balzer, N. Golman, and D. Wile, "On the transformational implementation approach to programming," *Proceedings 2nd International Conference on Software Engineering*, pp. 337-344 (1976).
3. F. L. Bauer, "Programming as an evolutionary process," *Proceedings 2nd International Conference on Software Engineering*, pp. 223-234 (1976).
4. J. M. Boyle and T. J. Harmer, "A practical functional program for the CRAY X-MP," ANL Preprint MCS-P159-0690, Argonne National Laboratory, Argonne, Illinois (July 1990).
5. J. M. Boyle, "Abstract programming and program transformation - an approach to reusing programs," *Software Reusability*, Vol. I, ed. Alan J. Perlis, ACM Press, Addison-Wesley Publishing Company, Reading, Massachusetts pp. 361-413 (1989).
6. J. M. Boyle, "Concrete syntax and semantics of TAMPR transformations," Preprint, Argonne National Laboratory, Argonne, Illinois (Sept. 1981).
7. J. M. Boyle, "Towards automated synthesis of linear algebra programs," *Production and Assessment of Numerical Software*, ed. M. A. Hennell and L. M. Delves, Academic Press, New York pp. 223-245 (1980).
8. R. M. Burstall and J. A. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM* 24(1), pp. 44-67 (Jan. 1977).
9. T. E. Cheatham and B. Wegbreit, "A laboratory for the study of automatic

- programming," *AFIPS Conference Proceedings*, SJCC 40, pp. 11-21 (1972).
10. A. Church, *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, New Jersey (1941).
 11. V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy, "A structure-oriented program editor: A first step towards computer assisted programming," *International Computing Symposium*, pp. 113-120 (1975).
 12. D. P. Friedman and M. Felleisen, *The Little Lisper*, 2nd ed., Science Research Associates, Inc., Chicago (1986).
 13. K. W. Hopkins, *Computer Program Transformation*, preprint, Greenville College Library, Greenville, Illinois (1983).
 14. K. W. Hopkins, "An informal introduction to parallel processors", Preprint MCS-P320-0892, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois (August 1992).
 15. J. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Massachusetts (1977).
 16. R. C. Waters, "The programmer's apprentice: Knowledge based program editing," *IEEE Transactions on Software Engineering* 8(1), pp. 1-12 (Jan. 1982).

DATE

FILMED

9/14/94

END