REFERENCES

[1] R. Balzer, N. Goldman, and D. Wile, "On the transformational implementation approach to programming" in *Proc. 2nd IEEE Int. Conf. Software Eng.*, 1976, pp. 337-344.

[2] R. Burton, "Semantic grammar: A technique for efficient language understanding in a limited domain," Ph.D. dissertation, Dep. Inform. Comput. Sci., Univ. California, Irvine, 1976.

[3] T. E. Cheatham, G. H. Holloway, and J. A. Townley, "Program refinement by transformation," in *Proc. 5th IEEE Int. Conf. Software Eng.*, 1981, pp. 430-437.

[4] P. Freeman, "Reusable software engineering: Concepts and research directions," in *Proc. ITT Workshop on Reusability in Programming*, Sept. 1983, pp. 2-16.

[5] L. Gonzalez, "A domain language for processing standardized tests," Master's thesis, Dep. Inform. Comput. Sci., Univ. California, Irvine, 1981.

[6] D. Kibler, J. M. Neighbors, and T. A. Standish, "Program manipulation via an efficient production system," *SIGPLAN Notices*, vol. 12, no. 8, pp. 163-173, 1977.

[7] J. M. Neighbors, "Software construction using components," Ph.D. dissertation, Dep. Inform. Comput. Sci., Univ. California, Irvine, Tech. Rep. TR-160, 1980.

[8] J. M. Neighbors, "Draco 1.1 manual," Dep. Inform. Comput. Sci., Univ. California, Irvine, Tech. Rep. TR-156, 1980.

[9] R. Prieto-Diaz and J. M. Neighbors, "Module interconnection languages: A survey," Dep. Comput. Inform. Sci., Univ. California, Irvine, Tech. Rep. TR-189, 1982.

[10] T. A. Standish, "PPL-An extensible language that failed," Center Res. Comput. Technol., Harvard Univ., Cambridge, MA, Rep. 15-71, 1971.

[11] T. A. Standish, D. Harriman, D. Kibler, and J. M. Neighbors, "The Irvine program transformation catalogue," Dep. Inform. Comput. Sci., Univ. California, Irvine, Tech. Rep., 1976.

[12] S. Sundfor, "Draco domain analysis for a real time application: The analysis," Dep. Inform. Comput. Sci., Univ. California, Irvine, Tech. Rep. RTP 015, 1983.

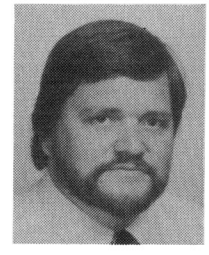
[13] —, "Draco domain analysis for a real time application: Discussion of the results," Tech. Rep., Dep. Inform. Comput. Sci., Univ. California, Irvine, RTP 016, 1983.

[14] F. Tonge and L. Rowe, "Date representation and synthesis," Dep. Inform. Comput. Sci., Univ. California, Irvine, Tech. Rep. TR-63, 1975.

[15] R. C. Waters, "The programmer's apprentice: Knowledge based program editing," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 1-12, Jan. 1982.

[16] D. S. Wile, "Program developments: Formal explanations of implementations," *Commun. ACM*, vol. 26, pp. 902-911, Nov. 1983.

[17] W. A. Woods, "Transition network grammars for natural language analysis," *Commun. ACM*, vol. 13, pp. 591-606, Oct. 1970.



James M. Neighbors (S'78-S'80-M'80-M'81) received the B.S. degree in computer science in 1974, the B.A. degree in physics in 1974, and the Ph.D. degree in computer science in 1980, all from the University of California, Irvine.

He was an Adjunct Assistant Professor in Computer Science at the University of California, Irvine, from 1980 to 1983. Currently, he is specializing in understanding the structure and aiding the development of existing large (500 K lines and up) software systems at the CXC Corporation, Irvine.

Dr. Neighbors is a member of the Association for Computing Machinery, the IEEE Computer Society, and the American Association for Artificial Intelligence.

Program Reusability through Program Transformation

JAMES M. BOYLE AND MONAGUR N. MURALIDHARAN

***Abstract*—How can a program written in pure applicative LISP be reused in a Fortran environment? One answer is by automatically transforming it from LISP into Fortran. In this paper we discuss a practical application of this technique—one that yields an efficient Fortran program. We view this process as an example of abstract programming, in which the LISP program constitutes an abstract specification for the Fortran version. The idea of strategy—a strategy for getting from LISP to Fortran—is basic to designing and applying the transformations. One strategic insight is that the task is easier if the LISP program is converted to "recursive" Fortran, and then the recursive Fortran program is converted to nonrecursive standard Fortran. Another strategic insight is that much of the task can be accomplished by converting the program from one canonical form to another. Developing a strategy also involves making various implementation decisions. One advantage of program transformation methodology is that it exposes such decisions for examination and review. Another is that it enables optimizations to be detected and implemented easily. Once a strategy has been discovered, it can be implemented by means of rewrite-rule transformations using the TAMPR program transformation system. The transformational approach to program reuse based on this strategy has a measure of elegance. It is also practical—the resulting Fortran program is 25 percent faster than its compiled LISP counterpart, even without extensive optimization.**

Manuscript received August 1, 1983; revised May 14, 1984. This work was supported by the Applied Mathematical Sciences Subprogram (KC-04-02) of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

J. M. Boyle is with the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439.

M. N. Muralidharan is with the Department of Computer Science, University of Kentucky, Lexington, KY 40506.

0098-5589/84/0900-0574$1.00

Index Terms—Abstract programming, canonical forms, optimization, program transformation, pure applicative LISP, rewrite rules, stepwise refinement, TAMPR.

I. Introduction

YOU have written a moderately large program in pure applicative LISP (3200 lines, 56 functions). It is highly recursive. It is thoroughly debugged and has been in use for several years. You have received numerous requests for the system of which the program is a part. You have decided to make that system available as part of a larger package for which Fortran has been chosen as the transportable implementation language. The program must run compiled in order for its use to be practical.

What can you do?

Should you rewrite the program by hand in Fortran? If you do, you must, of course, contend throughout the entire rewrite with Fortran's lack of support (if not outright antagonism) for recursion and for the list data type. In all probability, these shortcomings of Fortran are the reasons the program was written in LISP in the first place. Moreover, as you may already have realized, you would also have to contend with debugging again. Debugging is a task always to be avoided, especially if it has already been done once! In this case, the lack of support for lists and recursion in Fortran will make it all the more difficult.

The alternative is to render the program into Fortran by some automatic means. Can you use program transformations—rules that repeatedly rewrite small fragments of the program—for this task?

II. Transforming LISP to Fortran

We faced exactly this problem in making the TAMPR program transformation system [11] transportable to a wide range of machines. Its major component, the transformer, is written in LISP. Although there is a LISP interpreter available in Fortran, the LISP F3 system [32], we know of no LISP-to-Fortran compiler.

Because of our interest in program transformation, we decided to use a "bootstrapping" approach—to use TAMPR to transform itself from LISP into Fortran. Such an approach would enable us to reuse the LISP program for the TAMPR transformer, in the sense that the Fortran version would be mechanically derived from it. But, even using transformations, the problems of implementing both recursion and the list data type still remain. Why, then, is transforming the program into Fortran attractive when rewriting it in Fortran is not?

There are at least three answers to this question. Perhaps the most important is that the transformation from LISP to Fortran is an automatic process rather than a manual one. If the program were rewritten in Fortran by hand, even using the present LISP program as a guide, errors would almost certainly creep in during the transcription. On the other hand, if correctness-preserving transformations are used to rewrite it automatically, the transformed program will retain all the reliability of the original one. For the TAMPR system, this advantage is significant because the transformer program is thoroughly tested. Over the years that it has been in use, it has correctly applied millions of transformations and has been used to produce widely used software, including the programs in the LINPACK package [20] for solving systems of linear equations. We would not wish to throw such reliability away lightly.

The second answer is that program transformations provide significant help in overcoming the problems posed by the lack of support for recursion and for the list data type in Fortran. In the Fortran version of the program, the implementation of these two features must be explicit. Their implementation is also global—it pervades the entire program. Virtually every statement contains details of implementing lists or recursion, and most statements contain a mixture of details of both. The difficulty with rewriting the program by hand is that both of these kinds of detail must be dealt with simultaneously. However, as you will see, the tasks of writing the transformations that implement lists and those that implement recursion can be completely separated. This separation enables us to think about each of the two problems in isolation.

The third answer is that, by using transformations to rewrite the program from LISP into Fortran, we can at the same time optimize the resulting program. Were we to rewrite the program in Fortran by hand, we might be reluctant to attempt these optimizations because the task of implementing recursion and lists is already very demanding intellectually.

In the remainder of this paper, we show how program transformation enables us to attain reliability, intellectual manageability, and high performance in reuse of a program. To do so, we describe how we have organized the transformation from LISP into Fortran, and we point out some interesting aspects of program transformations along the way.

Organizing a problem for solution by means of program transformations consists mainly of devising a suitable "strategy for transformation"—in this case, a step-by-step understanding of how the Fortran program evolves from the LISP one. Before discussing strategy, however, let us consider some properties of the type of LISP program we intend to transform, and what we mean by program transformation.

A. The LISP Program as an Abstract Program

The LISP program that we set out to transform to Fortran is not, perhaps, a typical one. It is written in pure applicative LISP.

When we say that the TAMPR transformer program is written in (almost) pure applicative LISP (PAL), we mean it is written in the subset of the language in which evaluating a function has no side-effects. (We say *almost* pure applicative LISP because the transformer does use the functions *read*, *print*, and *gensym*, which do have side-effects.) Specifically, the program does not use *prog* nor *setq*, and functions refer only to their parameters and to local λ-variables. Under this assumption, no variable is ever reassigned a new value. Thus, complicated data-flow analysis is not needed to determine what value a particular use of a variable represents in such a program. Each use of a variable is simply a shorthand for the value of the expression to which it is bound.

Formally speaking, when a program has the pure applicative property, the rules of Church's λ-calculus [19] can be used as the basis for transformations that manipulate it. That is, the transformation rules can freely (except for taking care to avoid name clashes) replace instances of an evaluation of a function by instances of a λ-variable, after binding that λ-variable to the result of evaluating the function. Similarly, they can replace

instances of a λ-variable by instances of its corresponding function evaluation.

In part because the transformer program has the pure applicative property, we call it an *abstract program*—one that is closer to the problem specification than is a conventional program, and one that omits many implementation details. In addition, the LISP program is abstract with respect to the Fortran one in that the LISP program contains no explicit details of how lists and recursion are implemented, while the Fortran one does. As we shall see, abstract programs are particularly well suited to manipulation by means of program transformations. (The relationship between abstraction and transformation is discussed further in [8], [9].)

We shall see more about how to take advantage of the abstractness of a pure applicative LISP program in the discussion of the strategy for transformation in Section II-C. First, however, we consider what we mean by program transformation.

B. Transformation as Automated Refinement

When we speak of "program transformation," we mean a process that makes a large number of relatively small alterations to a program—alterations that do not destroy the meaning of the program. The new program thus *evolves* from the old. In contrast to monolithic processes such as compilation, program transformation permits access to the program at intermediate steps in its evolution. Such access can be useful, as you will see, for discovering potential optimizations and for carrying them out.

The program that results from applying a transformation to some given program can also be thought of as a *refinement* of that program. In a sense, the transformed program represents the original one with some particular *implementation decision* (represented by the transformation) incorporated into it. The idea of constructing a program by *stepwise refinement* is discussed in [45]. Here, however, we emphasize carrying out the refinement automatically using transformations.

In order to carry out a substantial refinement, such as the one from LISP to Fortran—and carry it out automatically—we must identify the implementation decisions that motivate the transformations that implement it. Some of these decisions are ones that obviously must be made, such as how lists are to be represented, how a reference to a parameter of a function is to be represented, etc. Others are more subtle—the need for them emerges only after considering the consequences of other implementation decisions. Simplifications and optimizations are in this latter class.

Once we have begun to identify the implementation decisions, we can write one or more transformations that incorporate each into a program. Of course, we intend that each such transformation (or small group of transformations) be *correctness preserving*—that any property that can be proved about the program before it applies can still be proved about the program after it applies. Even when no properties have been proved about the original program, we then at least know that applying such transformations yields a program that does no worse than the original program does.

When carrying out refinements using the TAMPR program transformation system, we typically collect the transformations arising from a closely related group of implementation decisions into a *transformation set*. We shall consider some of the major transformation sets that are needed for the LISP to Fortran refinement in the next section.

If we wish the set of transformations that carries out a refinement to apply automatically, either we must guarantee that the order in which its transformations apply makes no difference, or we must have some means to control the order in which they apply. The TAMPR system uses a mixture of these two approaches.

1) Each transformation (or small group of transformations) is intended to be correctness-preserving—if it applies and rewrites a fragment of the program, the program remains correct. Thus, regardless of the order in which the transformations apply, the resulting program will be correct.

2) As you will see, TAMPR itself, using the program being transformed, defines an order of application for the transformations in a set. That order can be used to guarantee not only that the transformations produce a correct program but also that they *make progress toward a goal.*

TAMPR does not apply a set of transformations directly to the program being transformed. Rather, it applies them to the parse tree of that program. The application takes place during an *exhaustive postorder* traversal of the nodes of the parse tree. By postorder traversal, we mean that before any transformations from a set are applied to a given node of the parse tree, that set is (recursively) applied to its subnodes. And by exhaustive traversal, we mean that whenever a transformation from a set does apply to a node in a parse tree, that set is reapplied to the parse tree resulting from the replacement, using an exhaustive postorder traversal. Only when the parse tree is completely traversed without any transformations from the set applying does the process terminate.

This ordering rule permits us to write each transformation using the assumption that *all fragments that its pattern will match in the program will have been fully transformed by the set of transformations of which it is a part.* Typically, a set of transformations is designed to achieve the goal of placing the program in some type of *canonical form* (for example, one in which all algebraic expressions are fully multiplied out). The ability to make this assumption provides the basis for an inductive proof that each transformation in a set (and hence the set as a whole) makes progress toward the canonical form. We call this inductive rule for proving properties of sets of transformations *transformation induction.*

An interesting property of this exhaustive postorder rule that TAMPR uses to apply sets of transformations is that the order of application of the transformations is primarily under control of the program being transformed. Such a rule permits a set of transformations to achieve its goal while at the same time adapting itself to the particular program to which it is being applied. Thus, it contributes greatly to the economy and power of automated program transformation.

C. Strategy for Transformation

With this background, we can now consider how to organize the transformational refinement of a program from LISP to Fortran. The process is one that starts from the LISP program

written in terms of lists and recursion and gradually evolves it into a Fortran program written in terms of arrays, indexes, and go-to statements. The idea of small changes that underlies program transformation encourages us to break the problem of transforming LISP to Fortran, which appears rather daunting at first, into a sequence of major steps—an application of the "divide-and-conquer" metastrategy.

The first major insight is to divide the problem into two steps: conversion of the LISP program to *recursive* Fortran, and conversion of the recursive Fortran to executable Fortran 66 or 77. Suddenly, the problem does not appear quite so daunting as before!

Implementing recursion in Fortran has been discussed briefly by Larmouth [27], [28]. We have chosen a similar approach, which can be described as follows.

- Replace all local variables in a function subprogram by references to positions in an array representing a stack.
- Replace all formal parameters of a function by references to the stack.
- Replace the actual parameters passed to a function when it is called by statements that place their values on the stack.
- Return the function result on the stack.
- Replace the use of the value of a function by a reference to the result of the function on the stack.
- Embed the code for all functions into a single program unit, beginning the code for each function with a label and ending it with a computed go-to statement (whose labels are those of the statements that follow the calls to that function).
- Replace each call to a function with two statements: one that stacks an index (for a computed go-to statement) specifying the label following that call and one that transfers to the label of the function body.

These steps correspond roughly to steps in the transformation process—to sets of transformations. Thus we now have a further subdivision of the second major step of the LISP to Fortran transformation strategy.

The preceding steps also provide some guidance about how to divide and conquer the first major step of the strategy, transforming LISP to recursive Fortran. Clearly, they require replacing each function call in the recursive Fortran program by a sequence of statements. But, sequences of statements cannot be directly substituted for function calls in Fortran since function calls are operands of expressions. Thus, it would be useful for the recursive Fortran form of the program to have at most one call to a nontrivial function in each statement. (By nontrivial function we mean a function other than the LISP primitives, such as *car*, *cdr*, etc.)

1) Transformation to Prepared PAL: How can we manipulate the program to reflect this implementation decision—manipulate it so that it has only one function call per statement? Fortunately, this idea has a precursor in the λ-calculus: binding the result of a function evaluation to the λ-variable of a λ-expression. To see why creating a λ-expression is useful, consider the LISP subexpression in Fig. 1, in which one of the arguments of the function f is a nontrivial function evaluation of the function g. A subexpression equivalent to this one is that in Fig. 2, in which the result of evaluating the function g is bound to a λ-variable. Such a subexpression has a straightforward translation into Fortran (assuming that the name of the λ-variable is unique), which is given in Fig. 3.

```
...(f (g a b) c)...
```

Fig. 1. LISP expression with embedded function evaluation.

```
...((lambda (l) (f l c)) (g a b))...
```

Fig. 2. Embedded function evaluation bound to λ-variable.

```
l = g(a,b)
...f(l,c)...
```

Fig. 3. Fortran translation of λ-expression.

```
...(f ((lambda (l) l) (g a b)) c)...
```

Fig. 4. Start of transformation of expression of Fig. 1.

Of course, we are using program transformation methodology—with its central idea of making only small changes—so we do not go directly from the expression of Fig. 1 to that of Fig. 2. Instead, we use several steps.

To carry out the first step of the process, we write a refinement transformation that simply replaces each nontrivial function evaluation that is an argument to another function evaluation by the equivalent λ-expression. (This transformation thus embodies the implementation decision that such a function evaluation should ultimately appear in a statement by itself.) The result of this rather pointless-appearing—but nevertheless correct—exercise is shown in Fig. 4.

To carry out the rest of the process, we write other transformations that enlarge the scope of such λ-expressions (as necessary) by distributing them out of the outer function evaluation in order to obtain, for this example, the expression of Fig. 2. The correctness of such transformations is, of course, justified by properties of Church's λ-calculus. (These transformations codify the implementation decision that in order for the λ-expressions to be *useful*—not just *correct*—they should have this particular scope.)

Preparing for the removal of recursion from the Fortran program by inserting λ-expressions in the LISP program in this way is useful because it can be done with complete reliability at the applicative level. That is, the transformations that create the λ-expression can be shown to be correct because they implement identities in the λ-calculus, and appealing to the λ-calculus is, in turn, justified by the assumption that the program being transformed is written in pure applicative LISP.

We call this language level, in which all nontrivial function evaluations have been bound to λ-variables, "prepared pure applicative LISP."

2) Transformation to Canonicalized PAL: Transforming directly from PAL to prepared PAL is still a fairly big step, however. Having at least one intermediate language level is useful. We call it "canonicalized pure applicative LISP" because it is a canonical form (albeit a weak one) for LISP programs. The motivation for introducing it is to simplify all subsequent transformations by reducing the number of special cases with which they must cope. Some of the canonicalizations are:

1) generating a Fortran function header for each LISP function definition and assigning the LISP expression in the function body to the Fortran function identifier;

```
(cond (p1 e1) (p2 e2) (t e3))
```

Fig. 5. LISP conditional expression.

```
use (e1) if (p1) otherwise
   use (e2) if (p2) otherwise (e3)
```

Fig. 6. TAMPR extended-Fortran conditional expression.

```
((lambda (x)
   ((lambda (y)
      (cons x (cons y nil))
   )
      (read)
   )
)
   (read)
)
```

Fig. 7. LISP function with order of evaluation specified.

```
((lambda (x y)
   (cons x (cons y nil))
)
   (read) (read)
)
```

Fig. 8. LISP function with order of evaluation unspecified.

2) expanding LISP "nlambda" functions—those with a variable number of arguments, such as *and, or*, and *list*—into their definitions in terms of repeated application of LISP functions;

3) renaming duplicate λ-variables so that the name of every λ-variable is unique within a function;

4) converting multiple-variable λ-expressions into nested single-variable ones;

5) converting LISP conditional expressions such as that shown in Fig. 5 into nested TAMPR extended-Fortran conditional expressions such as that shown in Fig. 6.

The first of these canonicalizations paves the way for the ultimate translation to recursive Fortran, by introducing the assignment that will be transformed as discussed in Section II-E. The second, fourth, and fifth reduce the variety of LISP constructs in the program by eliminating those that can be defined in terms of others.

The third canonicalization is perhaps the most important, for it permits later transformations to enlarge the scope of λ-expressions freely. Several sets of transformations, including those that transform to prepared PAL, contain transformations that change the scope of λ-expressions. Without the third canonicalization, each such transformation would have to check whether changing the scope causes a name clash and, if so, change one of the clashing names. Instead, it is simpler to place the program in a canonical form that guarantees that name clashes cannot occur. (Of course, later transformations must be written so that they do not destroy the canonicalization. Thus, a transformation that introduces new λ-expressions must maintain this property by, for example, generating λ-variable names that differ from all other names in the program.)

a) Implementation Decisions Cost Something: Every time we make an implementation decision—apply a transformation that makes the program more concrete—we eliminate some possible versions, or *realizations*, of the program. In this respect, the consequences of making the implementation decision to convert multiple-variable λ-expressions to nested single-variable ones are interesting to consider. We decided to make this canonicalization because it simplifies later transformations. This decision is not without its cost, however, for it sacrifices parallelism in the program.

In LISP, the order of evaluating the arguments of a multiple-variable λ-expression is not defined. Therefore, if a multiprocessor computer were at hand, such arguments could be evaluated in parallel. However, once a multiple-variable λ-expression has been converted to a nest of single-variable ones, an order of evaluation has been fixed, and it is no longer easy to take advantage of parallelism. Thus, should we wish to run the transformed program on a parallel computer, we might wish to reconsider the decision to apply this particular canonicalization. (A similar discussion applies to imposing an order of evaluation on function arguments, which the transformations for recursion elimination do.) In the absence of parallelism, however, applying the canonicalization is essentially cost-free, and doing so provides the benefit of reducing the number of cases to be considered in later manipulations involving λ-expressions.

b) Implementation Decisions and Preserving Correctness: Transformations that carry out implementation decisions have another interesting property that can be illustrated by the transformation that converts multiple-variable λ-expressions to nested single-variable ones. As is the case for all the transformations we discuss here, this transformation preserves the correctness of the original program. By *preserves the correctness* of the original program, we mean that *any property that we can prove about the original program, we can still prove about the transformed program.* But the converse of this statement—that any property that we can prove about the transformed program, we can prove about the original program—is not true. Thus, the transformed program is not *equivalent* to the original.

In general, a program that has been transformed contains additional information not in that program before transformation. Thus, there are some properties that can be proved about the transformed program that cannot be proved (or even stated) about the original program. In the case of this canonicalization transformation, the transformed program specifies a particular order for the evaluation of the λ-arguments. And properties that depend on that order can be proved about it after transformation, but not before. Of course, if we were working with absolutely pure functional LISP, no such properties would exist, but since we admit the "impure" functions *read, print*, and *gensym* they can be formulated. For example, for the LISP expression in "canonical" form illustrated in Fig. 7, we can prove that, if it is given as input the list (*a*) followed by the list (*b*), then it will produce the list ((*a*) (*b*)). For the program in Fig. 8, which uses a multiple-variable λ-expression, we can prove no such property—we cannot prove which of the lists ((*a*) (*b*)) and ((*b*) (*a*)) it will produce.

This situation—being able to state and prove more properties about a program after transformation than before—is typical. In fact, the ability of transformations to add information representing implementation decisions is the reason we say the transformed program is more *concrete* than the original. In addition, it shows how an abstract program in which many things are incompletely specified can be transformed into a con-

crete one in which virtually everything is fully specified. Finally, this ability can be used to explain why deriving several different concrete programs from a single abstract one is possible. For every transformation that represents an implementation decision, others could be written that reflect alternative ones. Each of these transformations could be used in its own transformational refinement, leading ultimately to a concrete program, which in all likelihood would differ from those produced by the others.

c) Implementation Decisions and Reversibility: Transformations that codify implementation decisions, such as the one to canonicalize λ-expressions, are frequently irreversible. The information they add to the program cannot readily be removed because proving that it is nonessential may not be possible. For example, given a program using nested single-variable λ-expressions (as in Fig. 7) it is impossible to tell *from the program alone* whether the fixed order of the evaluation of the λ-expressions has been used to prove any properties about the program. Therefore, we must assume that the order has been used and that transforming the program to multiple-variable λ-expression form (as in Fig. 8) would not preserve its correctness.

Most computer programs—the ones written every day to solve scientific and commercial problems—are concrete ones. As such, each embodies literally thousands of implementation decisions that are not dictated in any way by the specification of the problem to be solved. The fact that concrete programs have a plethora of such *possibly*, but not *provably*, irrelevant properties makes them difficult to modify, extend, adapt, and transport. On the other hand, abstract programs contain (almost by definition) only such information as is necessary to show that they solve the problem for which they were written. Therefore, modifying, extending, adapting, and transporting them is much easier than it is for concrete programs. These issues, and especially their implications for the reuse of programs on different machines, are discussed further in [7] and [8]. The role of abstract programming in raising the level of proofs about computer programs is discussed in [46, ch. 12].

3) Preparing the LISP Program for Transformation: One more step in the conversion of LISP to recursive Fortran remains to be mentioned—the initial step of the process, in which an editor script is used to convert the LISP program into a single large expression in the TAMPR extended-Fortran language. This conversion consists primarily of moving opening parentheses to the right over function names and inserting commas between function arguments. Such changes convert the function notation of LISP into a form that is parsable in the TAMPR extended-Fortran grammar, so that the process of applying transformations can begin. The effect of this editing on a LISP program is illustrated in Figs. 10 and 11 in Section II-D.

4) Language Levels: As we have seen throughout the preceding discussion of strategy, the process of transforming LISP to Fortran can be thought of as moving the program through a sequence of levels of language, beginning at the pure applicative LISP level and ending at the executable Fortran level. When we discuss language levels, however, we do not mean that the levels are separate languages. Rather, the language levels are all part of the same language—in our case, the TAMPR extended-Fortran language.

In this respect, the transformational approach to abstract programming presents a sharp contrast to approaches based on "abstract machines" and machine-independent intermediate languages, such as P-code (see, for example, [42]). Such approaches typically require that the abstract program be converted completely to the level of the intermediate language, whereupon the abstract program is thrown away.

Instead, in the transformational approach it is often the case that some parts of the program are at one level while others are at another. For example, as discussed in Section II-C-2 (and illustrated by the program in Figs. 12 and 13 in Section II-D), when the program is at the canonicalized pure applicative LISP level, part of each function body is in assignment form and part is in λ-expression form. This situation is typical. In general, it is useful to be able to have different parts of the program at different levels. The reason is that doing so permits information needed for optimization to be communicated easily to the stage of the process when it can be used effectively. If the entire program is forced to be at the same level simultaneously, information needed for optimization may be irretrievably lost, for reasons similar to those that make transformation irreversible, as discussed in Section II-C-2-c.

A language that encompasses a wide range of levels, from the descriptive to the imperative, has been christened a *wide-spectrum language* by Bauer. He discusses the role of wide-spectrum languages in program transformation briefly in [4]. Further discussion can be found in [5]. Wide-spectrum languages are of interest for program transformation precisely because they do permit a program to be at many levels simultaneously.

5) Strategy Summary: In this section we have discussed an overall strategy that we can employ to transform a program in pure applicative LISP into one in Fortran. The most important point is that we must factor—break up—the problem into a sequence of intellectually manageable steps—ones that we can hope to implement correctly. To do so, we can use the divide-and-conquer approach. Of course, we must factor the problem into intellectually manageable steps whether we use transformations or not. But, using transformations, with their emphasis on small, correctness-preserving changes, makes doing so mandatory.

Other important points that we have touched upon are the importance of canonicalization, the cost of implementation decisions, the irreversibility of some transformations, and the role of a wide-spectrum language with its many levels.

The language levels identified in the discussion of strategy are summarized in Fig. 9. In practice, several of these levels are split—refined—into two or more levels. The levels depicted in Fig. 9 are the conceptually important ones, however.

D. An Example Program

In this section, we give an example to make the strategic ideas of the preceding section more concrete. We illustrate what a simple program looks like at the various stages depicted in Fig. 9. This program, consisting of two functions, traverses a tree and produces a list of its terminal nodes. The LISP form of the program is shown in Fig. 10.

```
    Pure Applicative Lisp program
—>
    Pure Applicative Lisp, expressed in
        extended-Fortran syntax
—>
    Canonicalized Pure Applicative Lisp
—>
    Prepared Pure Applicative Lisp
—>
    Recursive Fortran, one function call
        per statement
—>
    Recursive Fortran, no dynamic local
        variables (local variables stacked)
—>
    Recursive Fortran, no parameters
        (parameters stacked)
—>
    Nonrecursive Fortran, executable
```

Fig. 9. Language levels between LISP and Fortran.

```
(defun treeleaves (lambda (tree)
    (cond
        ((null (cdr tree)) (list (car tree)))
        (t (treelistleaves (cdr tree)))
    )
))

(defun treelistleaves (lambda (treelist)
    (cond
        ((null treelist) nil)
        (t
            (append
                (treeleaves (car treelist))
                (treelistleaves (cdr treelist))
            )
        )
    )
))
```

Fig. 10. Program for a tree flattener.

```
expressions
function  (treelistleaves, body  (args (treelist),
 cond  (cc(null  (treelist), nil),
  cc(true(dummy), append  (treeleaves  (car  (treelist)),
   treelistleaves  (cdr  (treelist)) ) ) ));
end;
```

Fig. 11. Edited, TAMPR extended-Fortran program.

```
integer function treelistleaves ( treelist ) ;
preamble declare integer treelist ;
    endpreamble ;
endpreamble
treelistleaves .=1.
    use nil if ( null ( treelist ) ) otherwise
        append ( treeleaves ( car ( treelist ) ) ,
            treelistleaves ( cdr ( treelist ) ) ) ;
return ;
end ;
```

Fig. 12. Pure applicative LISP, canonicalized.

From this point on, we concentrate on the second function, treelistleaves. It is the more interesting of the two because it has nontrivial function calls in the call to append. The first step is to edit the program to convert it to the TAMPR extended-Fortran syntax. The edited form of the treelistleaves function is shown in Fig. 11.

The result of transforming this program to canonicalized PAL is shown in Fig. 12. Recall that at this stage, Fortran function headers and the assignment of the body expression to the function identifier are introduced. The assignment symbol .=1. is a *marker*—a synonym for = that indicates that this is the first assignment to the variable treelistleaves. This marker plays a role in optimizing the number of temporary variables in the final program, as discussed in Section III.

In the next step, this program is transformed to prepared PAL, as shown in Fig. 13. The transformations create lambda expressions for nontrivial function evaluations.

```
integer function treelistleaves ( treelist ) ;
preamble declare integer treelist ;
    endpreamble ;
endpreamble
treelistleaves .=1.
    use nil if ( null ( treelist ) ) otherwise
        lambda ( body ( args ( a00004 ) ,
            lambda ( body ( args ( a00005 ) ,
                apply ( compose ( append ) ,
                    toargs ( a00004 , a00005 ) ) ) ,
                apply ( compose ( treelistleaves ) ,
                    toargs ( cdr ( treelist ) ) ) ) ) ,
            apply ( compose ( treeleaves ) ,
                toargs ( car ( treelist ) ) ) ) ) ;
return ;
end ;
```

Fig. 13. Pure applicative LISP, prepared.

```
integer function treelistleaves ( treelist ) ;
preamble declare integer treelist ;
    endpreamble ;
endpreamble
if ( null ( treelist ) ) then ;
    treelistleaves .=1. nil ;
end else
    treelistleaves .=1. apply ( compose ( treeleaves ) ,
        toargs ( car ( treelist ) ) ) ;
    block ;
        declare integer a00005 ;
        end ;
        a00005 .=1. apply ( compose ( treelistleaves ) ,
            toargs ( cdr ( treelist ) ) ) ;
        treelistleaves = apply ( compose ( append ) ,
            toargs ( treelistleaves , a00005 ) ) ;
    end ;
end ;
return ;
end ;
```

Fig. 14. Prepared PAL, λ- and conditional-expressions coded.

```
integer function treelistleaves ( treelist ) ;
preamble declare integer treelist ;
        integer a00005 ;
    end ;
endpreamble
if ( treelist .eq. nil ) then ;
    treelistleaves .=1. nil ;
end else
    treelistleaves .=1. apply ( compose ( treeleaves ) ,
        toargs ( car ( treelist ) ) ) ;
    a00005 .=1. apply ( compose ( treelistleaves ) ,
        toargs ( cdr ( treelist ) ) ) ;
    treelistleaves = apply ( compose ( append ) ,
        toargs ( treelistleaves , a00005 ) ) ;
end ;
return ;
end ;
```

Fig. 15. Recursive Fortran, one function call per statement.

The next language level indicated in Fig. 9 is recursive Fortran, with one function call per statement. The program actually passes, however, through an intermediate level (as mentioned in Section II-C-5). At this level, conditional statements and assignments with local declarations of variables have been introduced, but the local declarations have not yet been moved to the beginning of the function. The program at this level is illustrated in Fig. 14. (The disappearance of the λ-variable a00004 is explained in Section III.)

The transition to the recursive Fortran level of Fig. 9 is completed by a set of library transformations that merge declarations that occur in parallel blocks and move them to the specification part of the Fortran function definition, and by transformations that introduce Fortran equivalents for the basic LISP predicates (*null, atom*, etc.). At this level, the program would be executable in an implementation of Fortran that supported recursion (if such things as if-then-else statements were converted to Fortran if statements and go-to statements, and if apply-compose were converted to function calls). The program at the recursive Fortran level is shown in Fig. 15.

The remaining stages of the transformation to standard Fortran are concerned with recursion removal. The transforma-

```
integer function treelistleaves ( treelist ) ;
preamble declare integer treelist ;
   end ;
endpreamble
jp = jp - 1 ;
if ( jp .le. ip ) call stkerx ;
if ( treelist .eq. nil ) then ;
   treelistleaves .=1. nil ;
end else
   treelistleaves .=1. apply ( compose ( treeleaves ) ,
      toargs ( car ( treelist ) ) ) ;
   stack ( jp ) .=1. apply ( compose ( treelistleaves ) ,
      toargs ( cdr ( treelist ) ) ) ;
   treelistleaves = apply ( compose ( append ) ,
      toargs ( treelistleaves , stack ( jp ) ) ) ;
end ;
jp = jp + 1 ;
return ;
end ;
```

Fig. 16. Recursive Fortran, no dynamic local variables.

```
subroutine treelistleaves ;
jp = jp - 3 ;
if ( jp .le. ip ) call stkerx ;
if ( stack ( jp + 1 ) .eq. nil ) then ;
   stack ( jp + 2 ) .=1. nil ;
end else
   stack ( jp - 2 ) = car ( stack ( jp + 1 ) ) ;
   call treeleaves ;
   stack ( jp + 2 ) = stack ( jp - 1 ) ;
   stack ( jp - 2 ) = cdr ( stack ( jp + 1 ) ) ;
   call treelistleaves ;
   stack ( jp ) = stack ( jp - 1 ) ;
   stack ( jp - 2 ) = stack ( jp + 2 ) ;
   stack ( jp - 3 ) = stack ( jp ) ;
   call append ;
   stack ( jp + 2 ) = stack ( jp - 1 ) ;
end ;
jp = jp + 3 ;
return ;
end ;
```

Fig. 17. Recursive Fortran, no parameters.

tions that carry them out embody implementation decisions about how recursion will be represented.

The transformations of the first stage place local variables of functions in a stack frame and place the stack frame on a stack. Thus, at this level (shown in Fig. 16) the language permits recursive calls to functions, but these functions may not use local variables. As a result, the local variable a00005 of Fig. 15 has been replaced by a reference to stack (jp) in Fig. 16.

At the next level, the subset of the language used by the program permits only recursive calls to parameterless subroutines—all aspects of recursion have been implemented except the stacking of return points from recursive calls. The transformations to this level place the parameters and returned function values into the stack frame with any local variables, both in function definitions and function calls. Then they convert function definitions to subroutine definitions and replace function calls by parameterless subroutine calls. The resulting program is shown in Fig. 17.

The transition to the final level, executable nonrecursive Fortran, is a fully global process, requiring a transformation that applies to the entire list of function definitions. The implementation decision embodied in this set of transformations is to use go-to statements to implement recursive calls and computed go-to statements to implement recursive returns. In order to do so, all the subroutine definitions must be merged into a single monolithic subroutine because, in Fortran semantics, labels are local to the subroutine in which they occur. As the transformations fold each individual subroutine into the monolithic one, they also generate a label for the first statement of its body and labels for all points of return from calls to it. The label on the body is used in the go-to statement that replaces each call to that subroutine. All the return point labels

```
      subroutine treeleaves
      stack(ip-1) = 1
      go to 20
   10 continue
      return
c     beginning of treeleaves
   20 continue
      ip = ip + 1
      jp = jp - 2
      if (jp .le. ip) call stkerx
      if (cdr(stack(jp)) .ne. nil) go to 30
         stack(jp-2) = car(stack(jp))
         stack(jp-3) = nil
         call kons
         stack(jp+1) = stack(jp-1)
      go to 50
   30 continue
         stack(jp-2) = cdr(stack(jp))
         stack(jp-1) = 1
         go to 60
   40    continue
         stack(jp+1) = stack(jp-1)
   50 continue
      jp = jp + 2
      irlab = stack(ip)
      ip = ip - 1
      go to (10,80), irlab
c     beginning of treelistleaves
   60 continue
      ip = ip + 1
      jp = jp - 3
      if (jp .le. ip) call stkerx
      if (stack(jp+1) .ne. nil) go to 70
         stack(jp+2) = nil
      go to 100
   70 continue
         stack(jp-2) = car(stack(jp+1))
         stack(ip-1) = 2
         go to 20
   80    continue
         stack(jp-2) = stack(jp-1)
         stack(jp-2) = cdr(stack(jp+1))
         stack(jp-1) = 2
         go to 60
   90    continue
         stack(jp) = stack(jp-1)
         stack(jp-1) = stack(jp+2)
         stack(jp-3) = stack(jp)
         call append
         stack(jp+2) = stack(jp-1)
  100 continue
      jp = jp + 3
      irlab = stack(ip)
      ip = ip - 1
      go to (40,90), irlab
      end
```

Fig. 18. Nonrecursive Fortran 77, executable.

for the subroutine are collected in a list that is used in the computed go-to statement that replaces its return statement. This list is also used to generate an index for the label of each point of return; this index is stacked just prior to the "call." (The return index is stacked at the opposite end of the stack from the variable and argument frame, in order to maintain compatibility with the LISP F3 system [32], whose list and atom representation, garbage collector, and read and print routines we use to provide list-processing support for the executable Fortran program.)

The results of these transformations are shown in the program in Fig. 18. This program is shown in executable, rather than structured, Fortran form, and it includes the code resulting from both of the functions in the original LISP program in Fig. 10. TAMPR generates the executable Fortran form of a program from the structured form by applying transformations and a set of formatting instructions that convert structured constructs such as if-then-else statements and do-while loops to their representations in terms of logical-if and go-to statements. (The program in Fig. 18 is written in Fortran 66 except for the use of subscripted variables as subscripts, which is permitted in Fortran 77 but not in Fortran 66. If a Fortran 66 program is desired, the transformations can be modified to assign such subscripts to temporary variables.)

Against this background, we discuss in the next section how transformations can be used to implement the transition from one language level to another.

```
.sd.
   <var>"1" =
      lambda ( body
         ( args ( <var>"2" ) ,
           <expr>"1" ) ,
        <expr>"2" ) ;
==>
   block ;
      declare
         integer <var>"2" ;
      enddeclare ;
      <var>"2" = <expr>"2" ;
      <var>"1" = <expr>"1" ;
   end ;
.sc.
```

Fig. 19. λ-expression coding transformation.

```
f = lambda(body(args(lv1),g(x,lv1)),f(x));
```

Fig. 20. A λ-expression.

```
block;
   declare integer lv1;
   enddeclare;
   lv1 = f(x);
   f = g(x,lv1);
end;
```

Fig. 21. Transformed λ-expression.

```
.sd.
   <var>"1" =
      use <var>"2"
      if ( <expr>"1" )
      otherwise <var>"3" ;
==>
   if ( <expr>"1" ) then ;
      <var>"1" = <var>"2" ;
   end else
      <var>"1" = <var>"3" ;
   end ;
.sc.
```

Fig. 22. Conditional expression coding transformation.

E. Some Representative Transformations

In the preceding sections, we have developed a general strategy for transforming pure applicative LISP into Fortran, and we have illustrated its application to a simple example program. Now we consider how to write the transformations that enable the TAMPR system to carry out on actual programs the manipulations dictated by the strategy.

As you follow the progress of the example program through the levels depicted in Figs. 10-18, you will notice that in all of the early levels it looks basically like a LISP program—more correctly, like a collection of Fortran functions with LISP expressions for their bodies. In the later stages—from the recursive Fortran level to the executable Fortran level—it looks much like a typical Fortran program. A dramatic change occurs, however, in passing from the prepared pure applicative LISP level (Fig. 13) to the recursive Fortran level (Fig. 15). It is in this transition that the program leaves the functional, expression-oriented domain of LISP and enters the imperative, statement-oriented one of Fortran.

1) Coding Transformations: The set of transformations that effect the first part of the transition from the program in Fig. 13 to that in Fig. 14 is an interesting one to study in more detail. Moreover, it is representative of the sets of transformations that are used at other levels.

The coding of λ-expressions and conditional expressions at the prepared pure applicative LISP level involves implementing such expressions in terms of Fortran assignment and conditional statements. The TAMPR transformation for coding λ-expressions is shown in Fig. 19. This rule describes how to code an assignment statement in which the right-hand-side is a λ-expression. It consists of a *pattern*—the part between **.sd.** and the arrow (==>)—and a *replacement*—the part between the arrow and **.sc.** The pattern matches an assignment statement that has a variable on the left (as, in fact, do all assignments) and a λ-expression on the right. When the pattern matches such an assignment, the replacement describes how to assemble a statement to substitute for it. The replacement of this transformation assembles an extended-Fortran block that declares the λ-variable (<var>"2") and assigns to it the value of the actual argument expression (<expr>"2") in the original λ-expression; it follows this assignment by one that assigns the λ-body expression to the original left-hand side variable. For example, given the program fragment shown in Fig. 20, the transformation in Fig. 19 applies to it and produces the program fragment shown in Fig. 21.

Of course, either or both of the assignment statements created by the replacement of this transformation may now be of the form described by the pattern of this rule, or that of one of the other rules discussed later. The exhaustive postorder rule that TAMPR uses to apply sets of transformations (see Section II-B) ensures that rules such as this one are applied until no more matching instances remain in the program. That is, when a new instance of the pattern of some transformation is created in a replacement, the rule insures that eventually that transformation will be applied to the new instance.

A transformation similar to the one for coding λ-expressions codes an assignment statement in which the right-hand side is a TAMPR extended-Fortran conditional expression. This transformation is shown in Fig. 22.

In addition to the preceding pair of transformations, the set of transformations for coding λ-expressions and conditional expressions also contains another, similar pair that code such expressions when they occur in the test of an if statement. A transformation set consisting of these four rules is sufficient to describe the coding at this level. However, in practice we use a set consisting of twelve transformations in order to carry out certain optimizations, as discussed in Section III.

The set of transformations just described bring the program from prepared PAL to the level depicted in Fig. 14. The remainder of the transformation to the recursive Fortran level is accomplished by two sets of transformations. One codes LISP predicates (for example, null (treelist) in Fig. 14) as Fortran relations (treelist .eq. nil). These transformations, of course, depend on the particular data representation chosen for lists. The other set of transformations (originally developed for use in another context) moves the local declarations introduced by the coding transformations to the head of the function definition. In the process, it merges variables declared with disjoint scope in order to minimize the number of temporary variables.

2) Coding as Canonicalization: Although at first glance you may not realize it, the set of transformations that code λ-expressions and conditional expressions are actually manipulating the program into a new canonical form. The effect of these transformations is to distribute assignment over the λ-operator and the conditional operator.

The canonical form produced by these transformations can

be easily understood by analogy. Think of assignment as multiplication, the λ-operator as addition, and the conditional operator as subtraction. Then, these transformations carry out the analog of converting algebraic expressions to fully multiplied-out form. Thus the transformation from Fig. 13 to Fig. 14 is analogous to rewriting the expression $a \times (-b + c + d)$ as $-a \times b + a \times c + a \times d$.

The strong connection between sets of transformations and canonical forms—the connection that sets of TAMPR transformations *define* canonical forms—is an interesting topic for further investigation. From this point of view, the problem of transforming LISP to Fortran can be solved by converting the program from one canonical form to another, just as some algebraic problems can be solved by multiplying out a factored expression and then factoring it in a different way. Thus, it is the conversion of one canonical form to another that "compiles" LISP to Fortran.

It appears to us that any process of program transformation, including traditional language compilation, can be viewed in this way. *This point of view is of important practical interest, for it can be used to guide the demonstration of the correctness of such processes.*

3) Transformations for Other Levels: The sets of transformations between the other language levels are similar to those that code λ-expressions and conditional expressions. They are described in further detail in [30].

F. Practical Results

The strategy for transforming pure applicative LISP programs into Fortran ones discussed in Section II-C and illustrated in Section II-D is implemented by 90 major correctness-preserving transformation rules of the type discussed in Section II-E. These transformations are divided into 20 independent transformation sets.

These transformations have been used to convert a 1300-line, 42-function subset of the LISP program for the TAMPR transformer to Fortran. (This subset implements all the major functionality of the transformer except for the generation of new program identifiers.) To convert this program, TAMPR applies the transformations 10 000 times to produce a Fortran program of about 3000 lines. Applying the transformations requires no manual assistance (interaction); the entire process is automatic.

The Fortran program produced by the transformations executed correctly (after circumventing some Fortran compiler bugs) the first time it was run, producing output identical to the LISP version. This Fortran program runs 25 percent faster on a DEC VAX 11/780 than it does in compiled Franz Lisp (Opus 38.26) on that machine. A number of well-understood optimizations that are made by the Franz Lisp compiler (for example, removal of tail recursion) have not yet been implemented as part of the LISP-to-Fortran transformations. We expect that further modest improvements in execution speed will result from implementing these optimizations.

The transformation process itself is not particularly efficient. The conversion of the 1300-line subset of the transformer required about four hours with the transformer running in compiled Franz Lisp on a VAX 11/780, or about 1.4 s per transformation applied. The longest single step (about 1.3 h) is the last transition of Fig. 9—the transition that implements calls to recursive parameterless subroutines. This process is fully global, since throughout the program calls to each function must be replaced by go-to statements and labels. The transformations for this step are not written in an efficient form. For clarity, they are written in a way (described in Section II-D) that makes m passes over the program where m is the number of functions in the original LISP program. Since m is roughly proportional to the length of the program, this set of transformations is effectively $O(n^2)$ in a measure n of the length of the program; it could be written to be $O(n)$ by generating and collecting return-point labels for all functions in a single pass.

The total time for all the steps of recursion removal is 2.5 h. Of the remaining 1.4 h, 0.5 h is spent in collecting LISP quoted constants (another global process not discussed here), and 0.5 h is spent performing the refinement from canonicalized pure applicative LISP to prepared pure applicative LISP. In contrast, applying the set of coding transformations discussed in Section II-E-1 (including the optimization discussed in Section III)—in which the individual transformations are not global—requires only 0.1 h.

We regard the time of four hours to transform this program from LISP to Fortran as slow, but not impossibly so. It is certainly fast—and inexpensive—compared to carrying out the task by hand.

A question about the generality of these techniques that frequently comes up is: Could the techniques and transformations we discuss here be used for LISP programs that are not written in pure applicative LISP? Actually, the strategy and transformations make only weak use of the pure applicative property—a use that does not differ significantly from that made by a typical LISP compiler. For example, none of the transformations replaces multiple instances of a function evaluation (common subexpressions) by a single λ-variable, which would represent a strong use. However, the transformations do create λ-expressions with a single use of the λ-variable and then enlarge their scope. Such manipulations can change the order of evaluation of functions and, hence, could change the behavior of a program in which side-effects of functions were permitted. However, these changes would be no worse than those made by some existing LISP compilers, in which the order of evaluation of the actual arguments of multiple-variable λ-expressions (and also of the arguments of functions) is different in compiled code than in interpreted code.

Thus, while we have not yet examined the problem in detail, we believe that the transformations described here could be extended easily to handle a class of nonapplicative LISP programs. This class consists of those programs that use *prog* and *setq* in a relatively disciplined way—ones that use only variables that are local to a function definition or global to all function definitions. In addition, we have outlined, but not implemented, a transformational approach to handling the use of lexically scoped variables in general.

III. Advantages of Program Transformation

The program transformation approach to implementing abstract programs offers a number of advantages, some of

```
lv1 = p(a,b,c);
if (lv1 .eq. nil) then;
    lv2 = nil;
else;
    lv2 = f(x,y,a);
end;
lv3 = g(x,y,lv2);
```

Fig. 23. Multiple temporary variables.

```
lv1 .=1. p(a,b,c);
if (lv1 .eq. nil) then;
    lv1 = nil;
else;
    lv1 = f(x,y,a);
end;
lv1 = g(x,y,lv1);
```

Fig. 24. Reuse of a temporary variable.

which we summarize and illustrate in this section. For each we give an aphorism as a mnemonic aid.

The fact that transformations work best when they are used to make large numbers of small changes strongly encourages factoring the whole problem to be solved by the transformations into a sequence of small, intellectually manageable subproblems. Such small problems are easier to solve correctly than is the original monolithic problem. In turn, because the small tasks can be done correctly, you can show that the program produced by the transformations is correct. Examples of this advantage are woven throughout the preceding sections. *Small is beautiful.*

Using transformations exposes design decisions that you otherwise might make implicitly. Once these decisions are exposed, you can examine their consequences, as well as the consequences of their alternatives. An example of this advantage is discussed in Section II-C-2-a. *Look before you leap.*

Organizing implementation decisions by codifying them in transformations, and the resulting fact that they are incorporated sequentially into the program, exposes opportunities for optimization that you might otherwise overlook. For example, if you use the λ-expression coding transformation given in Fig. 19, it produces, at the recursive Fortran level, fragments of code that look like that shown in Fig. 23. In this example, three temporary variables, lv1, lv2, and lv3 are used where obviously one would suffice. Because the code is available for examination after this set of transformations applies, it is easy to consider an alternative implementation decision: to reuse program variables wherever possible when coding λ-expressions. A set of transformations that implement this decision (which is, in fact, the set used to produce the examples in Section II-D) produces the program of Fig. 24 instead of that in Fig. 23. Here the *marker* .=1. is used to indicate the assignment statement in which a variable is first assigned; obviously, the variable is available for reuse before the marked statement. The optimizing transformations are written to recognize an assignment statement followed by a first assignment to an available variable and to reuse the available variable in that situation.

What is interesting is that this optimization is not possible at earlier language levels because it is a property of assignment statements that cannot be reflected in the scope of λ-expressions. Hence it cannot be made earlier. *Haste makes waste.* Similarly, if we do not make the optimization as we introduce the assignment statements (by marking the first assignment to a variable), the optimization becomes difficult to make, because flow analysis is required. Hence this optimization should not be made later. *Opportunity knocks but once.* There is a summary for these two aphorisms. *To everything there is a season.*

This example also illustrates that making one optimization frequently enables others. It is silly to assign the value nil to lv1 in Fig. 24, for lv1 has just been tested and is known to have

```
lv1 .=1. p(a,b,c);
if (lv1 .ne. nil) then;
    lv1 = f(x,y,a);
end;
lv1 = g(x,y,lv1);
```

Fig. 25. Reuse of a temporary variable, simplified code.

that value. So it is clear that we should add the obvious optimization of not assigning a variable a value it already has. The necessary transformation is trivial to write, and using it ultimately produces the fragment shown in Fig. 25. This fragment saves one assignment and reduces the size of the program over that in Fig. 24. The important point here is that this optimization is not possible until the implementation decision to reuse variables has been taken. *Optimization begets optimization.*

IV. Related Work

In some sense, the history of the ideas of abstract programming and program transformation can be traced to the use of macro processors to raise the level of abstraction of assembly language programs. Later, macro processors were used with higher level languages and the abstract machine concept. However, macro processors do not "understand" higher level programming languages—they operate on a program in such a language as if it were an unstructured string of characters. Thus, macro processors offer little assistance in implementing transformations of programs correctly; in fact, typical macro implementations tend to be difficult to design and debug.

True program transformation systems "understand" syntactic constructions in higher level languages—identifiers, variables, expressions, statements, and the like—and manipulate them reliably. They permit only well-formed constructs of the same syntactic type to be substituted for one another; thus they assist in writing transformations that are at least syntactically correct. The first such program tranformation system was implemented by Boyle in 1970 for the Algol 60 programming language [6]. It is a direct ancestor of the TAMPR system discussed here.

A. Other Program Transformation Systems

There are several other projects that emphasize abstract programming and production of efficient programs by transformation. One is Project CIP at the Technical University of Munich. An early discussion of this work can be found in [4], while a recent example appears in [13]; there are also numerous technical reports available from the University. Project CIP has emphasized the development of a wide-spectrum language, CIP-L, for abstract programming as well as a transformation system. An important feature of CIP-L is its precise formal definition, which is based on the algebraic theory of abstract data types. Another program transformation project is the GIST project at the Information Sciences Institute. Descriptions of this project can be found in [2], [3], [44].

This project has emphasized the development of a very high-level specification language (GIST) that accommodates problem-oriented notations implemented transformationally. An interesting feature of the project is the development of a program that automatically paraphrases GIST specifications in English. Burstall and his colleagues at the University of Edinburgh have studied transformations of recursion equations and recursive programs extensively [14], [22]. Darlington is continuing this work with emphasis on hardware implementations of transformations. Program transformation and abstract programming also play an important role in the extensible programming language ECL developed at Harvard; an early reference is [17]. From this work has evolved the current Harvard Program Development System (PDS) [16]. Other implemented transformation systems include that of Arsac [1] and the MENTOR system [21]. The original version of MENTOR was designed to transform Pascal programs. It has now been implemented with table-driven components so that it can, like TAMPR, be configured to operate on any language. MENTOR is currently being commercialized. The M.I.T. Programmer's Apprentice project of Rich and Waters [43] uses a generalized notion of transformation applied to "plans" to perform program synthesis.

Information on the current status of several of these systems can be found in the proceedings of a workshop on program transformation recently held in Munich [37]. In addition, this proceedings contains an extensive record of the participants' discussion of current issues in program transformation. Finally, a recent survey article [36] contains an excellent and extensive (193 entry) bibliography on program transformation topics.

B. Transformation and Optimization

A number of papers mention the important connection between abstraction, transformation, and optimization. Two early examples are "catalogs" of correctness-preserving program optimization transformations, the Irvine Program Transformation Catalog [40] and that of Loveman [29]. In Loveman's catalog there is also an interesting example of the use of transformations to optimize the implementation of a moderately abstract (approximately Pascal-level) program. Some of the transformations in Loveman's catalog have been implemented in automatic optimizers, but in hard-coded, rather than rewrite-rule, form.

At a somewhat higher level of abstraction are transformations of LISP programs. They have long been used in LISP compilers to optimize programs before compilation, most notably for recursion removal but also for other optimizations as well. A good discussion of such transformations appears in [12]. A LISP transformation system that "explains" the nature or purpose of the transformations it applies is discussed by Steele [41]. It is an outgrowth of a LISP transformation system constructed to optimize the implementation of abstract data types in the TAMPR transformer.

Paige and Koenig discuss a particularly elegant class of transformations for highly abstract programs [35] that go beyond optimization and contribute to the implementation of algorithms. These transformations extend the classical mathematical idea of computing numerical functions by finite differencing to optimizing the implementation of set-theoretic abstractions in programs written in the SETL language.

Cheatham, Holloway, and Townley also discuss specifying the implementation and optimization of abstract algorithms by means of transformations [16]. They stress that when the developer of an algorithm specifies new abstractions (abstract data types and abstract control structures), he must also be able to specify optimizations for them. Even when a sophisticated optimizing compiler is available, there are always optimizations that the programmer is aware of but that such an automatic compiler cannot discover.

We strongly concur with this observation. The ability to specify *problem-domain dependent* optimizations and implementations of abstractions simultaneously is one of the major advantages of using program transformations. Moreover, as we have pointed out in Section III, there is usually a point in the transition from abstract program to concrete implementation at which each such optimization is particularly simple to apply. Thus, even if an optimizing compiler were to discover such optimizations, it would have to expend significant additional resources to apply them.

C. Automated Versus Interactive Transformation

Program transformation systems and catalogs of transformations are frequently depicted as interactive program development tools. In that mode, they rely on the user to select which transformation to apply, to confirm whether an automatically selected transformation should be applied, or to confirm an applicability condition for a transformation. Even when applying a transformation that does not have applicability conditions, they may ask the user to determine whether applying that transformation at a particular point in the program is a "good idea." In such systems, the transformations themselves do not incorporate a *strategy* for achieving a goal. Systems that require interaction for each application of a transfromation (or a small group of transformations) have an obvious disadvantage—they are cumbersome to use, especially for tasks of the LISP-to-Fortran type, which require thousands of such applications.

To overcome this disadvantage, some systems provide elaborate control languages in which to express detailed specifications of the order and place of application of transformations in the program tree. Typical of such control specifications are instructions to move up or down in the tree from the point of application of a particular transformation, and instructions to try certain transformations next after a particular one applies. These systems suffer from a less obvious, but more serious, disadvantage than interactive ones: What if there is a program for which the instructions are incorrect or incomplete? This question manifests itself as a difficulty in demonstrating the correctness of sets of transformations and control instructions in such systems.

In contrast, the TAMPR system offers automated operation without an elaborate control language. Interaction is limited to the initial selection of which sets of transformations to apply. These selections correspond directly to the programmer's design decisions about the target realization that he wishes. In place of explicit control, TAMPR uses control that is largely im-

plicit, governed by the exhaustive postorder (bottom-up) sequencing rule discussed in Section II-B. This implicit sequencing leaves the order of application of transformations primarily under control of the program being transformed. Implicit sequencing, coupled with the use of transformations that individually (or in small groups) preserve correctness, makes it easy to see that a set of transformations operates correctly for all admissible programs.

In our experience, automated operation with implicit sequencing is a very satisfactory mode of operation. (See [10] for a brief description of applications of TAMPR in addition to the LISP-to-Fortran transformation described here.) Experience with another automated transformation system, the Harvard PDS [16], corroborates this observation. The PDS uses an implicit sequencing rule–exhaustive preorder (top-down)–similar to that used in TAMPR. This rule is presumably more efficient than the TAMPR exhaustive postorder rule, but it does not permit the use of the transformational induction rule for canonical forms (Section II-B), which the TAMPR rule does.

The PDS also provides a notation for *limiting the scope* of transformations. Limited-scope transformations are useful, for example, for expressing optimizations that are valid only within the scope of some optimization or implementation expressed by another transformation. An analogous facility is provided by the notation for *subtransformations* in TAMPR (see [11]).

Finally, we remark that the TAMPR exhaustive postorder sequencing rule is the same as that used for the equality-rewriting process called *demodulation* in automated reasoning systems (see [46]).

There are other program transformation systems that emphasize automated transformations. One is the RAPTS system [33], [34], used by Paige to apply the finite differencing transformations discussed earlier. RAPTS uses either rewrite rules or hard-coded rules with built-in guidance, depending on the application. An interesting feature of RAPTS is that it transforms not only the program, but also measures of the program's complexity and various invariants, all at the same time. The complexity measure and invariants are used to guide the transformation of an abstract program into a concrete one. The primary interaction required with RAPTS is to make a few implementation decisions early in the transformation process, although RAPTS can ask the user to confirm applicability conditions if necessary.

Automated program transformation also plays an important role in the program synthesis systems of Green, for example, in the PSI system discussed in [24]. This system contained an elaborate component to help evaluate alternative implementations for abstract constructs (for example, sets) and to select the ones most efficient for a particular program.

D. Rewrite-Rule Versus Hard-Coded Transformations

An automated transformation system for conventional languages (the example implementation is for Pascal) that is not rewrite-rule based is the GRAMPS system [15]. Instead of syntactic pattern matching, it uses a procedural language, or "metaprogramming" language, to describe transformations. Cameron and Ito state that the metaprogramming approach has decided advantages over approaches employing rewrite rules, especially for expressing complex transformations. However, to us these metaprograms seem more cumbersome and less clear than rewrite rules. In fact, Cameron and Ito present most of the transformations in their paper first in terms of rewrite rules and then give the metaprogram for them. (The examples given in [31] provide further illustrations of the potential incomprehensibility of nonrewrite-rule transformations.)

Cameron and Ito state several times that they favor the metaprogramming approach because rewrite rules are "incomplete" and because they are insufficiently powerful to handle such tasks as expression simplification. In contrast, we find that rewrite rules, at least as implemented in the TAMPR system, are complete and well suited to most program manipulation tasks. As far as the general question of completeness is concerned, TAMPR rewrite rules have been proved Turing-complete [6]. (This result is not surprising in view of the fact that TAMPR rewrite rules are based on Chomsky transformational grammars, which are Turing-complete [18].)

Of course, a proof of theoretical completeness does not address the question of sufficiency–not to mention convenience–in practical use. Let us consider some specific cases for which Cameron and Ito find rewrite rules to be insufficient. One is the need for infinite sets of rewrite rules to express certain optimizations (see [15, sect. 5.2]). The TAMPR system solves this problem within the context of rewrite rules by permitting rules that contain *dynamic patterns* and *dynamic replacements*–rules that specify how to compute an instance of the infinite set of rewrite rules that will match the particular program being transformed.

Another task for which Cameron and Ito believe rewrite rules to be insufficient is to express the implementation (evaluation) of applicability conditions for transformations. As an example, they cite the predicate *DeadEffectsQ*, which characterizes those expressions in a program that are free from side effects (see [15, sect. 5.1]). Of course, a single rule cannot express the evaluation of such a predicate, but a set of rules can. One approach is to write a set of rules that evaluate a complicated predicate and then insert a syntactic "marker" (like the .=1. mentioned in Section III) to indicate the result. A convenient marker is a renaming of the identity function where the name signifies that some property–in this case, being free from side-effects–is true of its argument. Then, rewrite rules that require this applicability condition need only explicitly match the inserted marker. (Of course, such markers temporarily complicate the appearance of the program being transformed. However, their explicit presence in the program can be an advantage, in contrast to implicit properties computed by some other mechanism. When a transformation fails to apply, the fact that markers are explicit facilitates figuring out why.)

Thus, in this approach, what appear to be nonsyntactic applicability conditions in a transformation are converted into sets of transformations each of which computes an applicability condition and represents the result syntactically with a marker. Then the applicability conditions in the original transformation can be replaced by purely syntactic checks for the appropriate markers. Once this approach is understood, it can be packaged neatly by extending the syntax of rewrite rules to

permit the transformations that compute applicability conditions to be embedded in the original transformation. TAMPR provides a **.where.** *clause* for this purpose. In many cases, using it permits one to dispense with the syntactic marker.

Based on the above observations, we favor using rewrite rules to express nearly all aspects of program transformation. Doing so avoids the complication of using different means for different tasks. And it fosters writing correct transformations because determining whether a rewrite rule preserves correctness is easier than determining whether a metaprogram does.

E. Abstract Programming

A number of authors have discussed abstract programming without regard to a specific transformation system. An early (and continuing) advocate of an abstract programming methodology for writing business programs is Jackson [25]. The transformations needed to realize efficient concrete programs in this methodology were to be carried out manually. Knuth discusses abstract programming in relation to the disciplined use of the go-to statement in [26]. Gerhart discusses the role of abstraction in simplifying proofs of program correctness in [23] (see also [46]). Finally, Scherlis and Scott give an elegant philosophical discussion of the potential of abstract programming in [39] (see also [38]). Further references on abstract programming can be found in [36].

V. Conclusions

A 1300-line subset of the TAMPR transformer program has been reused by transforming it from applicative LISP to Fortran. The resulting Fortran program is moderately large–3000 lines–and fairly efficient–25 percent faster than the compiled Franz Lisp version (see Section II-F). This example demonstrates that program reuse through program transformation is practical.

Approaching this task by means of program transformation encourages organizing it in a modular fashion. The general approach is the following.

1) Develop a strategy for solving the problem, by breaking it up into intellectually manageable steps and corresponding language levels.

2) Implement each of the steps as one or more sets of correctness-preserving program transformations that move the program from one language level to the next.

3) Apply the transformations to produce a correct program.

4) Examine this program at intermediate levels to determine whether further optimizations could be made.

An important point is that the transformation rules that carry out the steps codify implementation decisions and knowledge about programming. Frequently such decisions and knowledge, and hence the transformations, can themselves be reused in other contexts.

Other possibilities for reuse arise because each of the language levels in the spectrum defined by the strategy is a potential point of departure or port of entry for other conversions. For example, the recursive Fortran program might be executed directly, if a suitable compiler were available. Alternatively, it might serve as the point of departure for producing a Pascal program. Similarly, the conversion of other recursive languages to Fortran might enter the spectrum at the recursive Fortran level. This sharing of language levels again makes it possible to reuse a large fraction of the transformations in other, related applications.

Finally, we have discussed a number of ways in which transformations illuminate the underlying principles of programming. This increased understanding of computer programming–a process exposed here as largely a science, not an art–is perhaps the greatest benefit of the transformational approach to program reusability.

Acknowledgment

Over the years a number of people have contributed to the development of the TAMPR system and the transformations discussed in this paper. A version of the recursion removal transformations for recursive numeric programs was written by D. S. Dick. The set of transformations that merge declarations, which was reused in this application, was written by K. Wieckert. The TAMPR formatter was developed by K. Dritz with the assistance of C. Hauser. T. Harmer, D. Henderson, and B. Char worked on the TAMPR recognizer. K. Dritz, K. Hopkins, C. Metzger, P. Newton, M. Matz, B. K. Steele, B. Smith, S. Hague, and W. Barth have written transformations for applications of TAMPR. L. Wos provided comments on a draft of this paper. Finally, we wish to acknowledge P. Messina and the members of the Mathematics and Computer Science Division for their support of this work.

References

[1] J. J. Arsac, "Syntactic source to source transformation and program manipulation," *Commun. ACM*, vol. 22, pp. 43–54, Jan. 1979.

[2] R. Balzer, D. Cohen, M. S. Feather, N. M. Goldman, W. Swartout, and D. S. Wile, "Operational specification as the basis for specification validation," in *Theory and Practice of Software Technology*, D. Ferrari, M. Bolognani, and J. Goguen, Eds. Amsterdam, The Netherlands: North-Holland, 1983, pp. 21–49.

[3] R. Balzer, N. Goldman, and D. Wile, "On the transformational implementation approach to programming," in *Proc. 2nd Int. Conf. Software Eng.*, 1976, pp. 337–344.

[4] F. L. Bauer, "Programming as an evolutionary process," in *Proc. 2nd Int. Conf. Software Eng.*, 1976, pp. 223–234.

[5] F. L. Bauer, M. Broy, H. Partsch, and P. Pepper, "Report on a wide spectrum language for program specification and development," Tech. Univ. München, Munich, West Germany, Tech. Rep. TUM-18104, May 1981.

[6] J. M. Boyle, "A transformational component for programming language grammar," Argonne Nat. Lab., Argonne, IL, ANL-7690, July 1970.

[7] —, "Towards automatic synthesis of linear algebra programs," in *Production and Assessment of Numerical Software*, M. A. Hennell and L. M. Delves, Eds. New York: Academic, 1980, pp. 223–245.

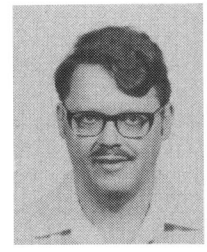
[8] —, "Program adaptation and program transformation," in *Practice in Software Adaptation and Maintenance*, R. Ebert, J. Lueger, and L. Goecke, Eds. Amsterdam, The Netherlands: North Holland, 1980, pp. 3–20.

[9] —, "Software adaptability and program transformation," in *Software Engineering*, H. Freeman and P. M. Lewis II, Eds. New York: Academic, 1980, pp. 75–94.

[10] —, "Lisp to Fortran–Program transformation applied," in *Program Transformation and Programming Environments* (NATO ASI Series, Vol. F8), P. Pepper, Ed. Berlin, Heidelberg, New York: Springer-Verlag, 1984, pp. 291–298.

[11] J. M. Boyle and M. Matz, "Automating multiple program realizations," in *Proc. MRI Symp., XXIV: Comput Software Eng.* Brooklyn, NY: Polytechnic, 1976, pp. 421–456.

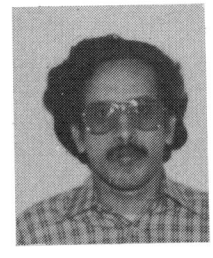
[12] R. A. Brooks, R. P. Gabriel, and G. L. Steele, Jr., "An optimizing complier for lexically scoped LISP," *ACM SIGPLAN Notices*, vol. 17, pp. 261–275, June 1982.
[13] M. Broy, and P. Pepper, "Combining algebraic and algorithmic reasoning: an approach to the Schorr-Waite algorithm," *ACM Trans. Programming Languages Syst.*, vol. 4, pp. 362–381, July 1982.
[14] R. M. Burstall and J. A. Darlington, "A transformation system for developing recursive programs," *J. Ass. Comput. Mach.*, vol. 24, pp. 44–67, Jan. 1977.
[15] R. D. Cameron, and M. R. Ito, "Grammar-based definition of metaprogramming systems," *ACM Trans. Programming Languages Syst. (TOPLAS)*, vol. 6, pp. 20–54, Jan. 1984.
[16] T. E. Cheatham, Jr., G. H. Holloway, and J. A. Townley, "Program refinement by transformation," in *Proc. 5th Int. Conf. Software Eng.*, pp. 430–437, 1981.
[17] T. E. Cheatham, and B. Wegbreit, "A laboratory for the study of automatic programming," *AFIPS Conf. Proc., SJCC*, vol. 40, pp. 11–21, 1972.
[18] N. Chomsky, "Three models for the description of language," *IRE Trans. Inform. Theory*, vol. IT-2, pp. 113–124, 1956; Reprinted (with corrections) in *Readings in Mathematical Psychology, Vol. II*, R. D. Luce, R. Bush, and E. Galanter, Eds. New York: Wiley, 1965.
[19] A. Church, *The Calculi of Lambda Conversion*. Princeton, NJ: Princeton Univ. Press, 1941.
[20] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *Linpack User's Guide*. Philadelphia, PA: Soc. Ind. Appl. Math., 1979.
[21] 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," in *Proc. Int. Comput. Symp.*, pp. 113–120, 1975.
[22] M. S. Feather, "A system for assisting program transformation," *ACM Trans. Programming Languages Syst. (TOPLAS)*, vol. 4, pp. 1–20, Jan. 1982.
[23] S. L. Gerhart, "Knowledge about programs: A model and case study," in *Proc. Int. Conf. Reliable Software*, Apr. 21-23, 1975, pp. 88–95.
[24] C. C. Green, "The design of the PSI program synthesis system," in *Proc. 2nd Int. Conf. Software Eng.*, 1976, pp. 4–18.
[25] M. A. Jackson, *Principles of Program Design*. London, England: Academic, 1975.
[26] D. E. Knuth, "Structured programming with GOTO statements," *ACM Comput. Surveys*, vol. 6, pp. 261–301, Dec. 1974.
[27] J. Larmouth, "Serious FORTRAN," *Software Practice and Experience*, vol. 3, no. 2, pp. 87–107, 1973.
[28] —, "Serious FORTRAN–Part 2," *Software Practice and Expereience*, vol. 3, no. 3, pp. 197–225, 1973.
[29] D. B. Loveman, "Program improvement by source to source transformation," *J. ACM*, vol. 24, pp. 121–145, Jan. 1977.
[30] M. N. Muralidharan, J. M. Boyle, and D. Smith-Dick, "Transformations for converting applicative Lisp into Fortran–Implementation notes," Argonne Nat. Lab., Argonne, IL, Tech. Rep., 1984, to be published.
[31] E. W. Myers, Jr. and L. J. Osterweil, "BIGMAC II: A FORTRAN language augmentation tool," *Proc. 5th Int. Conf. Software Eng.*, pp. 410–421, 1981.
[32] M. Nordstrom, LISP F3 Users Guide, Datalogilaboratoriet, Uppsala Univ., Uppsala, Sweden, June 1978.
[33] R. Paige, "Transformational programming–Applications to algorithms and systems," in *Proc. 10th ACM Symp. Principles of Programming Lanugages*, Jan. 1983, pp. 73–87.
[34] —, "Supercompilers–extended abstract," in *Program Transformation and Programming Environments* (NATO ASI Series, Vol. F8), P. Pepper, Ed. Berlin, Heidelberg, New York: Springer-Verlag, 1984, pp. 331–340.
[35] R. Paige and S. Koenig, "Finite differencing of computable expressions," *ACM Trans. Programming Languages Syst. (TOPLAS)*, vol. 4, pp. 402–454, July 1982.
[36] H. Partsch and R. Steinbrüggen, "Program transformation systems," *ACM Comput. Surveys*, vol. 15, pp. 199–236, Sept. 1983.
[37] P. Pepper, Ed., *Program Transformation and Programming Environments* (NATO ASI Series, Vol. F8). Berlin, Heidelberg, New York: Springer-Verlag, 1984.
[38] W. L. Scherlis, "Software development and inferential programming," in *Program Transformation and Programming Environments* (NATO ASI Series, Vol. F8), P. Pepper, Ed. Berlin, Heidelberg, New York: Springer-Verlag, 1984, pp. 341–346.
[39] W. L. Scherlis, and D. S. Scott, "First steps towards inferential programming," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-83-142, July 1983; also in *Proc. IFIP Congress 83*.
[40] T. A. Standish, D. C. Harriman, D. F. Kibler, and J. M. Neighbors, "The Irvine Program Transformation Catalog," Univ. California at Irvine, Jan. 1976.
[41] B. K. Steele, "An accountable source-to-source transformation system," Master's thesis, Dep. Elec. Eng. Comput. Sci., Massachusetts Institute of Technology, Cambridge, July 1980.
[42] W. M. Waite, *Implementing Software for Nonnumerical Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[43] R. C. Waters, "The programmer's apprentice: Knowledge based program editing," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 1–12, Jan. 1982.
[44] D. S. Wile, "Program developments: Formal explanations of implementations," *Commun. ACM*, vol. 26, pp. 902–911, Nov. 1983.
[45] N. Wirth, "Program development through stepwise refinement," *Commun. ACM*, vol. 14, pp. 221–227, Apr. 1971.
[46] L. Wos, R. Overbeek, E. Lusk, and J. Boyle, *Automated Reasoning: Introduction and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1984.



James M. Boyle received the B.S. degree (with highest distinction) in science engineering in 1964, the M.S. degree in mathematics in 1965, and the Ph.D. degree in applied mathematics in 1970 from Northwestern University, Evanston, IL.

From 1968 to 1969 he was part-time Instructor in the Department of Mathematics at Northwestern University. From 1967 to 1969 he was a Student Research Associate in the Applied Mathematics Division (A.M.D.), Argonne National Laboratory (A.N.L.), Argonne, IL. From 1969 to May 1974 he was Assistant Computer Scientist, A.M.D., A.N.L. Since 1974 he has been Computer Scientist in the Mathematics and Computer Science Division (formerly A.M.D.), A.N.L. His research interests include abstract programming, program transformation, multiple realizations of a program optimized for different hardware architectures, and automated reasoning.

Dr. Boyle is a member of the Amerian Association for the Advancement of Science, the Association for Computing Machinery, the Society of Industrial and Applied Mathematics, and Sigma Xi.



Monagur N. Muralidharan received the B.E. degree in electronics and communications engineering from Madras University, Madras, India, in 1972, and the M. Tech. degree in computer science in 1976 and the Ph.D. degree in computer science in 1983 from the Indian Institute of Technology, Kanpur, India.

From 1978 to 1981 he was Senior Research Assistant at the Indian Institute of Technology. In 1982 he was a Postdoctoral Appointee with the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL. Since January 1983 he has been Assistant Professor in the Department of Computer Science, University of Kentucky, Lexington. His research interests include program transformations, functional programming, programming languages and operating systems.

Dr. Muralidharan is a member of the IEEE Computer Society.