In Proceedings of the ACM Nat'l Conf. pp 509-516, ACM, 1976.

IMPROVING AND REFINING PROGRAMS BY PROGRAM MANIPULATION*

Thomas A. Standish, Dennis F. Kibler, and James M. Neighbors
Department of Information and Computer Science
University of California at Irvine
Irvine, California 92717

By manipulating programs to reorganize the way they compute their results, it is possible to improve program performance in certain desirable directions. Further, using certain laws of exchange, it is possible to transform concise, abstract, high-level programs into efficient, concrete, underlying representations. This paper explores the notion of using source-to-source transformations interactively as a basis for an approach to program improvement and refinement. Examples of program manipulation are presented that convey both the flavor of the approach and its requirements.

1. *Introduction*

This paper explores concepts in program manipulation. Programs are viewed as objects containing subexpressions that can be rearranged using laws of exchange that preserve program equivalence. These exchange laws are called *source-to-source transformations*.

The examples we shall study are aimed at two purposes: *program improvement* and *program refinement*. Often it becomes necessary to improve a given program so that it meets certain required operational constraints. These operational constraints may derive from the requirement to execute the program in restricted space, or from response-time requirements. In addition, economic factors associated with extensive repetitive executions of the program may make various efficiencies highly desirable. Thus, when efficiency counts, programmers must be able to select appropriate underlying representations, and to guide the production of efficient mechanical details. The achievement of such efficiencies may occur at the expense of legibility and clarity, and may require close control over low-level implementation descriptions.

By contrast, during initial stages of program design, or during later stages of documentation and maintenance, it is advantageous for programmers to deal with programs written in a high-level language free from contamination with low-level implementation details, and written in a concise, clear, well-structured style.

A given high-level program may possess many underlying concrete implementations, each with different performance properties. Different data representations may support the same abstract program intent and may be accompanied by different appropriate underlying program text to express required operations. *Program Refinement* is the process of transforming a high-level program into a concrete underlying implementation by "filling in details" with reference to some specified representation conventions.

* This work was supported by the National Science Foundation under Grant DCR75-13875.

High-level to low-level transformations correspond to the concept of stepwise refinement that has received much attention in the structured programming literature [1,2], while transformations that improve programs have been the conventional focus of much of the literature on program optimization [3,4,5]. Both sorts of transformations are basic to the programming process, and, thus, it is interesting to inquire whether both can be mechanized at a level more comprehensive than that seen in current practice.

The study of mechanical program transformation is part of a larger research effort aimed at extending the range of coverage of mechanical programming assistance from contemporary tools such as text-editors, compilers, debuggers, and syntax-checkers, to encompass new capabilities such as program verification [6,7,8,9], program synthesis [10,11,12], and mechanical programmer's apprentices [7,15].

Following Knuth [16], our particular approach is aimed at the development of an *interactive program manipulation system*.

> The programmer using such a system will write his beautifully structured but possibly inefficient program P; then he will interactively specify transformations to make it efficient. Such a system will be much more powerful and reliable than a completely automatic one [16, p. 283].

We see several potential advantages in this approach. First, a mechanical transformation that has been shown to preserve program equivalence [25], carries out reliably what otherwise would be subject to human error, and saves the programmer from having to think through the details each time.

Second, mechanical transformations may make it possible to experiement with a change in the underlying representations of programs where manually it might otherwise be prohibitively expensive or unreliable. Oftentimes, the extensive volume of low-level detail in contemporary large-scale programs makes consideration of an extensive representation change unthinkable. However, if low-level concrete details have been generated mechanically from a high-level original by mechanical refinement,

then shifting representations becomes a great deal more economical, and a new dimension of freedom is opened up for the programmer.

Third, coupled with some sort of performance evaluation mechanism [17], we can isolate and then transform interactively those sections of the program whose efficiency is most critical to overall performance. Experience shows [18] that oftentimes a preponderant portion of the execution time is spent in a relatively small portion of a program. Effort spent at program improvement yields the highest payoff when directed at such places. In an interactive program manipulation system, a programmer could focus the optimization process with a degree of control not now attainable in contemporary optimizing compilers.

Fourth and finally, a mechanical assistant that helps manipulate and improve programs is the embodiment of an extensive stock of ideas for program transformation and refinement that may well exceed the capabilities an average programmer might otherwise bring to the task. Such a system then tends to act as a "force multiplier" in the production of efficient, reliable programs.

This paper focuses on examples that illustrate program manipulations we seek eventually to mechanize, and that convey, more or less, the flavor of the overall approach and its requirements.

2. A Notation for Transformations

In many areas of mathematics, transformations are given as rules of exchange of the form P = Q. For example, in algebra, we might find the distributive law given in a form such as X(Y+Z) = XY + XZ; in logic, we might find a simplification law given as B∧(B∨C) = B; and in a table of integrals, we might find a rule such as $\int u\,dv = uv - \int v\,du$. Each of these exchange laws can be used to replace a given expression with an equivalent one.

Similarly, we can specify exchange rules for expressions used in programs. For instance, we might give a distributive rule involving conditional expressions such as:

x + (*if* b *then* y *else* z) = (*if* b *then* x+y *else* x+z)

To specify that a given exchange rule is intended for use in a preferred direction, we write P => Q. For example, writing B∧(B∨C) => B signifies our preference for replacing the more complex expression B∧(B∨C) with the simpler expression B, where possible.

Since computer programs often contain the equal sign (=), the use of the double shafted arrow (=>) helps avoid ambiguities in the specification of transformations on expressions. For example, writing

$$b = c = d \;\Rightarrow\; (b = c) \wedge (c = d)$$

seems preferable to writing

$$b = c = d = (b = c) \wedge (c = d) \quad .$$

For this reason, we indicate bi-directional exchange rules using a double-headed arrow (<=>) as in X(Y+Z) <=> XY+XZ.

Some transformations can be applied only under special conditions. For example, provided $X \neq 0$, we can apply the cancellation law XY = XZ => Y = Z. Sometimes, we state such conditions formally as part of a transformation rule, as in writing

provided $X \neq 0$: XY=XZ => Y=Z .

In this case, the condition (X≠0) is called an *enabling condition*.

Not all source-to-source transformations can be conveniently stated as exchange rules. Sometimes it is preferable to give a procedure for transforming a program into another. This condition usually applies when we cannot easily devise a simple syntactic "pattern" to match situations in which a transformation applies.

For example, one source-to-source transformation of interest consists of removing useless assignment statements, which assign values to "dead" variables (those which are never subsequently referenced in a program). This process is better expressed as a procedure with several steps than as an exchange rule on various forms of programs.

Such procedures are program manipulating programs. In this situation, programs constitute dual entities that do the manipulating and are themselves manipulated. This requires introducing appropriate notational conventions to distinguish between the objects and agents of actions. However, in this paper we refrain from treatment of this issue on the grounds that informality better suits our exposition.

3. General Enabling Conditions

The transformations we study are intended to preserve *program equivalence*. This implies that we can apply certain transformations only under certain enabling conditions. Three such enabling conditions are of such general applicability, however, that they deserve special mention. These are called *commutativity*, *freedom from side-effects*, and *invariance*.

Let F be a well-formed program fragment (i.e. a phrase in the grammar of the programming language at hand). The *input variables* of F, denoted In(F), are those that F reads and never writes, or that F reads first before writing. The *local variables* of F, denoted Loc(F), are those that F writes before reading, which are also not input variables of any other program fragment G, reachable from F. The *output variables* of F, denoted Out(F), are non-local variables of F which F writes.

Given two program fragments F and G, we wish to know when it is permissible to exchange their order of execution while preserving program equivalence, since certain program transformations implicitly change the order of certain constituent program fragments. We shall say that two such fragments are *commutative* if their order of execution can be interchanged without loss of program equivalence. It can be shown that F and G are *commutative* provided

$$\mathrm{Out}(F) \cap \mathrm{Out}(G) = \emptyset, \text{ and}$$

$$\mathrm{In}(F) \cap \mathrm{Out}(G) \cup \mathrm{In}(G) \cap \mathrm{Out}(F) = \emptyset \ .$$

Intuitively, this means that F and G commute if they write into disjoint sets of output variables, and if neither writes into variables which are inputs of the other. They may, however, share the same read-only variables, and they may use identically named local variables.

Whenever we are given a transformation that implicitly changes the order of execution of some of its constituent program fragments, the fragments whose order is changed must be *commutative*. This requirement is so pervasive that it is wasteful to mention it every time it applies, and we shall assume in the sequel that it always applies. For example, in the

transformation:

```
                                  if b then
a; if b then c else d  =>             begin a;c end
                                  else
                                     begin a;d end
```

the order of execution of a and b is implicitly changed, so a and b must be commutative. This would exclude transforming

```
x←3;if x>w then y else z
```

into

```
if x>w then
    begin x ← 3; y end
else
    begin x ← 3; z end
```

for instance.

If the program fragment F does not write into any non-local variables, then we say F is *side-effect free*.

definition: F is *side-effect free* provided

$$Out(F) = \emptyset .$$

For example, the McCarthy conditional transformation:

$$a \wedge b \Rightarrow \textit{if } a \textit{ then } b \textit{ else false}$$

requires b to be side-effect free in order to preserve program equivalence, in general.

We say that a program fragment F is *invariant* with respect to a program fragment G provided G does not write into any variables that F reads.

definition: F is *invariant with respect to* G,

$$\text{provided } Out(G) \cap In(F) = \emptyset .$$

Invariance is required as an enabling condition for transformations which remove invariants from loops. The following transformation requires invariance:

provided P is invariant with respect to R, and P and Q are side-effect free:

```
while P do                          while P do
 if Q then R else S      =>            begin
                                         while Q do R;
                                         S
                                       end
```

4. *Improving a Program to Run Nearly Six Times Faster*

As our first example, we show how to use a small set of source-to-source transformations to improve a program fragment which multiplies two N×N, upper-triangular matrices A and B. The elements of the product matrix C are given by the formula:

$$C[i,j] = \sum_{k=1}^{N} A[i,k] \times B[k,j] .$$

A corresponding program fragment to compute C is:

```
for i ← 1 step 1 until N do
   for j ← 1 step 1 until N do
     begin
       C[i,j] ← 0;                                   (1)
       for k ← 1 step 1 until N do
         C[i,j] ← C[i,j] + A[i,k]×B[k,j]
     end
```

Given that A and B are upper-triangular matrices, we can write:

```
A[i,k] = (if i ≤ k then A[i,k] else 0 )
B[k,j] = (if k ≤ j then B[k,j] else 0 )        (2)
```

We now wish to simplify the above program fragment with respect to the latter conditional expressions to se if we can eliminate products involving matrix elements which are zero. We shall use the following laws of program transformation:

1. *distribution on conditionals*
   (a) X <operator> (*if* Y *then* Z *else* W) =>
   (*if* Y *then* X<operator>Z *else* X<operator>W)
   (b) (*if* Y *then* Z *else* W) <operator> X =>
   (*if* Y *then* Z<operator>X *else* W<operator>X)
2. *arithmetic identity elimination*
   (a) X × 0 => 0 0 × X => 0
   (b) X + 0 => X 0 + X => X
3. *eliminating assignments by identity*
   (a) X ← X => *empty*
4. *eliminating the empty program*
   (a) *provided* S is a <statement>:
   S; *empty* => S & *empty*; S => S
   (b) *for* k ← a *step* b *until* c *do empty* => *empty*
5. *splitting the range of a loop*
   (a) *provided* j ≤ b:

```
for k ← a step 1 until b do
  if k≤j then X else Y

=> for k ← a step 1 until j do X;
   for k ← j+1 step 1 until b do Y
```

   (b) *provided* i-1 ≤ b:

```
for k ← a step 1 until b do
  if i≤k then X else Y

=> for k ← a step 1 until i-1 do Y;
   for k ← i step 1 until b do X
```

In order to preserve program equivalence, X and Y must be commutative in transformation 1.a, X must be side-effect free in transformations 2.a and 3.a, k must be local in transformation 4.b, and a, b, and c must be side-effect free in transformation 4.b.

To carry out our program improvement, we begin by transforming the product term A[i,k]×B[k,j] used in the assignment C[i,j] ← C[i,j] + A[i,k]×B[k,j] in the innermost loop of our original program fragment. We expand this product term by substituting the conditional expressions given in (2) above:

```
A[i,k] × B[k,j]
      ⇓(2)
(if i≤k then A[i,k] else 0)
× (if k≤j then B[k,j] else 0)
```

The latter expression is repeatedly transformed using distribution on conditionals and arithmetic identity elimination as follows:

```
                    ⇓ 1.a

(if k≤j then (if i≤k then A[i,k] else 0) × B[k,j]
     else (if i≤k then A[i,k] else 0) × 0)

                    ⇓ 2.a

(if k≤j then (if i≤k then A[i,k] else 0) × B[k,j]
     else 0)
```

$\Downarrow$ 1.b

```
(if k≤j then (if i≤k then  A[i,k]×B[k,j]
        else 0×B[k,j]) else 0)
```

$\Downarrow$ 2.a

```
(if k≤j then (if i≤k then  A[i,k]×B[k,j]
        else 0) else 0)
```

We can now replace the product term in C[i,j] ← C[i,j] + A[i,k]×B[k,j] with the latter result, and apply further transformations as follows:

```
C[i,j] ← C[i,j] +
  (if k≤j then  (if i≤k then A[i,k]×B[k,j]
     else 0) else 0)
```

$\Downarrow$ 1.a (applied four times)

```
(if k≤j then (if i≤k then
  C[i,j] ← C[i,j] + A[i,k]×B[k,j]
    else C[i,j] ← C[i,j] + 0 )
      else  C[i,j] ← C[i,j] + 0)
```

$\Downarrow$ 2.b (applied twice)

```
(if k≤j then (if i≤k then
  C[i,j] ← C[i,j] + A[i,k]×B[k,j]
    else C[i,j] ← C[i,j])
      else C[i,j] ← C[i,j])
```

$\Downarrow$ 3.a (applied twice)

```
(if k≤j then (if i≤k then
  C[i,j] ← C[i,j] + A[i,k]×B[k,j]
    else empty) else empty)
```

The assignment C[i,j] ← C[i,j] + A[i,k] × B[k,j] in the inner loop of the original program fragment (1) can now be replaced with the latter result, giving a new version of the inner loop:

```
for k←1 step 1 until N do
  if k≤j then
    (if i≤k then
       C[i,j]←C[i,j]+A[i,k]×B[k,j] else empty)
  else empty
```

Application of the loop splitting transformation 5.a to the latter text yields:

$\Downarrow$ 5.a

```
for k ← 1 step 1 until j do
  (if i≤k then C[i,j]←C[i,j]+A[i,k]×B[k,j]
      else empty);
for k ← j+1 step 1 until N do empty
```

The second of these split loops collapses and disappears by transformations 4.b and 4.a. The first may again be split using transformation 5.b.

$\Downarrow$ 4.b, 4.a, and 5.b

```
for k ← 1 step 1 until i-1 do empty;
for k ← i step 1 until j do
  C[i,j] ← C[i,j] + A[i,k] × B[k,j]
```

The first of these split loops now collapses and disappears using transformations 4.b and 4.a again:

$\Downarrow$ 4.b and 4.a

```
for k ← i step 1 until j do
  C[i,j] ← C[i,j] + A[i,k] × B[k,j]
```

We may now substitute this improved version of the inner loop back into the original program fragment to obtain the final form:

```
for i ← 1 step 1 until N do
  for j ← 1 step 1 until N do
    begin
    C[i,j] ← 0;
    for k ← i step 1 until j do                (3)
      C[i,j] ← C[i,j] + A[i,k]×B[k,j]
    end
```

Analysis of the running times of the program fragments (1) and (3) reveals that the inner loop of (1) executes N^3 times, while the inner loop of (3) executes $N^3/6 + N^2/2 + N/3$ times. Thus, while both fragments are of computational complexity $O(N^3)$, we see that as $N \to \infty$, there is a nearly sixfold reduction in the number of executions of the inner loop of (3) compared to that of (1). Further, for $N > 15$, there is at least a fivefold reduction, a worthwhile improvement.

Once one shows that the transformations used preserve program equivalence [25], then the sequence of manipulations used to produce an improved program also constitute a *proof* that the improved program is equivalent to the original. Hence, if the original program has been proved correct, the improved program must be correct also.

We see that the inner loop of the improved program (3) of the form:

```
for k ← i step 1 until j do
  C[i,j] ← C[i,j] + A[i,k] × B[k,j]
```

is never executed if i>j. Hence, the program fragment (3) initializes C[i,j] to 0 when i>j, but it never subsequently changes the value of C[i,j]. This implies that the sub-diagonal elements of the product matrix C are 0. What we have, then, is a *proof by program transformation* that the product of two upper-triangular matrices is upper-triangular.

At each stage of the sequence of program manipulations above, we applied program simplification to keep "intermediate program swell" within bounds. This is similar to the phenomenon of "intermediate expression swell" that occurs in algebraic manipulation systems [19].

5. Replacing Procedure Calls with Partially Evaluated Procedure Bodies

A technique that is often effective in optimizing a program is to replace a procedure call with a partially evaluated copy of the procedure body. This is particularly worthwhile if the procedure call sits inside a heavily travelled inner loop and the compiler is forced to generate relatively expensive parameter passing code (as can happen in Algol 60, for instance, when shifting lexicographic execution environments between the site of the call and the site of the procedure body for parameters called by name). In addition, when a constant is used as an actual parameter to a procedure call, and replaces a given formal parameter uniformly throughout the corresponding procedure body, program simplification techniques may yield substantial improvements. Loops may collapse, arms of conditional expressions may drop away, relational expressions may simplify, and arithmetic may be performed. Under such circumstances, the procedure body may be said to be *partially evaluated* [20,21]. A limiting case occurs when all parameters to a procedure call are constants, enabling complete evaluation of the procedure call, and replacement of the

call by the run-time value (assuming the call is side-effect free and returns a value).

For example, suppose we are given a procedure

```
procedure P(a,b,c); integer a,b,c;
  begin integer x;
    x ← r;
    if a>1 then Q(b-x) else
        while c>0 do
          begin
          R(a-b);
          c ← c - 1
          end
  end procedure
```

Consider, now, the following call on P, where S_1 and S_2 are surrounding statements:

```
S1;
P(1,x+1,b);
S2;
```

Renaming the local variable x inside the body of P to be y, in order to avoid naming conflicts with the identically named variable x used as an actual parameter to the call, and substituting actual parameters for formal parameters yields the following renamed, substituted version of the procedure body:

```
begin integer y;
  y ← r;
  if 1>1 then Q((x+1)-y) else
    while b>0 do
      begin
      R(1-(x+1));
      b ← b-1
      end
end
```

Now using simplifications such as (1>1) => *false*, *if false then* A *else* B => B, and 1-(x+1) => -x, the latter text simplifies to:

```
begin integer y;
  y ← r;
  while b>0 do
    begin R(-x); b ← b-1 end
end
```

The final simplified piece of text can be used to replace the original call, P(1,x+1,b).

Generally speaking, partial evaluation of a renamed, substituted procedure body proceeds *inside-out*. By this we mean that simplification is first applied at the deepest levels of nesting, and then proceeds outward to the levels of nesting immediately containing results previously simplified. A brief incomplete outline of this process is given in Figure 1 below. A great deal more care and precision is required to carry out "suitable systematic renaming" than we have indicated briefly here, and the transformations required to simplify program forms are much more extensive than the outline in Figure 1 indicates. These matters are dealt with in [22], a program transformation catalogue containing over 100 groups of transformations, and spelling out such processes as "suitable systematic renaming" in detail.

Figure 1.

Substituting Simplified Versions of Bodies of Procedures for Calls on Those Procedures

1. *Renaming*: Eliminate name conflicts by suitable systematic renaming.
2. *Substitution*: Substitute actual parameters from the procedure call into the procedure text in places indicated by the corresponding formal parameters. [This is for "call by name" parameter passing. Another policy must be followed for "call by value" and "call by reference".]
3. *Simplification:*
   - A. *Arithmetic on Constants*: do all possible arithmetic resulting from the substitution of constants for variables.
   - B. *Simplify Arithmetic Expressions*: using at least the following transformations:

     X + 0 => X, 1 × X => X, X + 1 => X,
     X × 0 => 0, X / 1 => X, etc.

   - C. *Simplify Relational Expressions*: e.g.
     (X-Y)<relation>0 => X<relation>Y
   - D. *Simplify Boolean Expressions*: e.g.

     ~ *true* => *false* A ∧ B => (*if* A *then* B
     ~ *false* => *true* *else false*)
     A ∨ *true* => *true* A ∨ B => (*if* A *then true*
     A ∨ *false* => A *else* B)
     A ∧ *true* => A (*if* ~A *then* B *else* C) =>
     A ∧ *false* => *false* (*if* A *then* C *else* B)

   - E. *Simplify Control Forms*: e.g.

     (*if true then* A *else* B) => A
     (*if false then* A *else* B) => B
     while false do S => *empty*
4. *Replacement*: replace the original call on the procedure with the simplified body resulting from step 3.

6. *Procedural Abstraction*

Replacing a procedure call with a partially evaluated procedure body saves execution time usually at the expense of program size. The inverse transformation, *procedural abstraction*, can be used to generate a saving in program space at the expense of execution time. Procedural abstraction consists of replacing a set of similar pieces of code $P_1, P_2, \ldots, P_n$ with calls on a procedure P, which is derived by generalizing the P_i ($1 \le i \le n$).

For example, in the computation of the Cosine of the angle between two 3-vectors v and w, one might find the code:

```
S ← 0;
for i← 1  step 1 until 3 do S←S+v[i]×w[i]
R ← 0;
for i← 1  step 1 until 3 do R←R+v[i]×v[i]    (4)
T ← 0;
for i← 1  step 1 until 3 do T←T+w[i]×w[i]
Result ← S/(R×T)
```

By recognizing the common loop, one can replace this code by the following:

```
real procedure Dot(x,y); real array x,y[1:3]
  begin integer i;
    Dot ← 0;
    for i ← 1 step 1 until 3 do                (5)
        Dot ← Dot + x[i]×y[i]
  end;
Result ← Dot(v,w)/(Dot(v,v)×Dot(w,w))
```

One way to construct a procedural abstraction of a set of program fragments $P_1,P_2,\ldots,P_n$ involves determining the *least common generalization* [23,24]. We say that F is *less general* than G (written $F \leq G$), if there exists a substitution θ such that $F = \theta G$. If F and G are exactly equal under some renaming of their respective variables, we say they are *alphabetic variants* (written $F \sim G$). G is a *least common generalization* of two program fragments F_1 and F_2 if both $F_1 \leq G$ and $F_2 \leq G$, and there exists no $G' \leq G$ such that $F_1 \leq G'$ and $F_2 \leq G'$, and such that G' is not an alphabetic variant of G. The least common generalization is unique up to renaming of variables.

Roughly speaking, we can find a least common generalization of $P_1,P_2,\ldots,P_n$ by superimposing their "parse trees" on top of one another and by identifying the largest common portion of the set of superimposed trees containing the root. At any place where the roots of two or more subtrees differ, we can substitute a new formal parameter, which takes on different subtrees as values. In this fashion, variables replace the smallest regions of superimposed code that differ, and the largest common superstructure is identified.

Let F be a program fragment, and let P be a "pattern". Let $F{=}{=}P$ denote the relationship "F is of the form P". An example of an algorithm that constructs least common generalizations is as follows:

procedure $\Gamma(P_1,P_2,\ldots,P_n)$; Construct the least common generalization of the program fragments $P_1,P_2,\ldots,P_n$.

[1] If all the program fragments are identically equal, $P_1{=}P_2{=}\ldots{=}P_n$, then the procedure terminates with their common value P_1 as a result.

[2] If there exists a binary operator β such that $P_i = x_i\ \beta\ y_i$ for all i ($1 \leq i \leq n$), then the procedure terminates with the value $\Gamma(x_1,x_2,\ldots,x_n)\ \beta\ \Gamma(y_1,y_2,\ldots,y_n)$

[3] If there exists a unary operator α such that $P_i = \alpha y_i$ for all i ($1 \leq i \leq n$), then the procedure terminates with the value $\alpha\ \Gamma(y_1,y_2,\ldots,y_n)$.

[4] If $P_i = A_i[e_i]$ for all i ($1 \leq i \leq n$), then the procedure terminates with the value $\Gamma(A_1,A_2,\ldots,A_n)[\ \Gamma(e_1,e_2,\ldots,e_n)\ ]$.

[5] If P_i = *if* b_i *then* c_i *else* d_i for all i ($1 \leq i \leq n$), then the procedure terminates with the value *if* $\Gamma(b_1,b_2,\ldots,b_n)$ *then* $\Gamma(c_1,c_2,\ldots,c_n)$ *else* $\Gamma(d_1,d_2,\ldots,d_n)$

[6] If P_i = *for* $v_i \leftarrow a_i$ *step* b_i *until* c_i *do* S_i for all i ($1 \leq i \leq n$), then the procedure terminates with the value *for* $\Gamma(v_1,v_2,\ldots,v_n) \leftarrow \Gamma(a_1,a_2,\ldots,a_n)$ *step* $\Gamma(b_1,b_2,\ldots,b_n)$ *until* $\Gamma(c_1,c_2,\ldots,c_n)$ *do* $\Gamma(S_1,S_2,\ldots,S_n)$.

[7] Otherwise, the result is an n-tuple $\langle P_1,P_2,\ldots,P_n\rangle$. In other words, if the P_i are not instances of some common program form, shown in steps [1] through [6], then the result is an n-tuple of ungeneralized program fragments.

As an example, we apply Γ to the three for-statements used in the code fragment (4) above:

last line of page

Γ(*for* i←1 *step* 1 *until* 3 *do* S←S+v[i]×w[i],
for i←1 *step* 1 *until* 3 *do* R←R+v[i]×v[i], =>
for i←1 *step* 1 *until* 3 *do* T←T+w[i]×w[i])

for i ← 1 *step* 1 *until* 3 *do*
<S,R,T> ← <S,R,T> + <v,v,w>[i]×<w,v,w>[i]

The final step in the generalization process is to replace distinct tuples with distinct new variables. Letting <S,R,T> → Dot, <v,v,w>→ x, and <w,v,w> → y, we get:

for i ← 1 *step* 1 *until* 3 *do* Dot ← Dot + x[i]×y[i]

The particular choice of variables here is that required to generate the for-statement in the generalization (5) of the for-statements used in (4) above.

7. Speeding Up Evaluation of Boolean Expressions

The source-to-source transformation studied in this section is carried out by two procedures which call each other recursively. Let $F(a_1,a_2,\ldots,a_n)$ be a Boolean expression over the side-effect free Boolean primaries $a_1,a_2,\ldots,a_n$ (for $n \geq 1$) using only the operators $\{\sim,\vee,\wedge\}$. Let Ψ be a transformation that removes Boolean constants that are operands of Boolean operators by repeated application of the following transformations:

a ∨ *true* => *true*	a ∧ *true* => a
a ∨ *false* => a	a ∧ *false* => *false*
true ∨ a => *true*	*true* ∧ a => a
false ∨ a => a	*false* ∧ a => *false*

~ *true* => *false*
~ *false* => *true*

Define Φ as follows:

provided $n \geq 2$:
$\Phi(F(a_1,a_2,\ldots,a_n))$
=> *if* a_1 *then* $\Phi(\Psi(F(true,a_2,\ldots,a_n)))$
else $\Phi(\Psi(F(false,a_2,\ldots,a_n)))$

whereas if $n=1$ *then*:

$\Phi(F(a_1))$ => $\Psi(F(a_1))$

The mapping Φ transforms a Boolean expression $F(a_1,a_2,\ldots,a_n)$ into a nested conditional expression in which no argument a_i is evaluated more than once in any given execution. Specifically, if the a_i are Boolean variables, the evaluation time is at most proportional to the number of distinct variables in $F(a_1,a_2,\ldots,a_n)$.

For example, let $F(a,b,c,d) = (a \wedge \sim b) \vee (d \wedge (a \vee c))$. Then,

$\Phi((a\wedge\sim b)\vee(d\wedge(a\vee c)))$
=> *if* a *then* (*if* b *then* d *else true*)
else (*if* d *then* c *else false*) .

Essentially the resulting code is a binary decision tree based on successive tests of the arguments $a_1,a_2,\ldots,a_n$, so we may need to use an exponential amount of code to achieve evaluation times that are at worst proportional to the number of arguments.

8. Refinement of Abstract Data Structures

Given an algorithm A written using set notation (e.g. using expressions such as $D \cap (B \cup C)$, or *if* $X \in Y$ *then* $C \leftarrow C \cup \{X\}$), we may wish to utilize one of many possible underlying representations for sets, and we may attempt to map A onto an underlying concrete program with respect to the chosen representation. Here, we exemplify what might happen if the expres-

sion X∈Y were to be mapped onto underlying program text with respect to different choices of representations.

First suppose X is a character and that Y is a set of characters. Suppose further that we choose to represent Y by a list. We might assert formally:

(1) Let (X) be a (Character)
(2) Let (Y) be a (Set of (Character)s)
(3) Represent (Y) by a (List)

Now we apply the following deductive assertion to statement (3):

if (the Representation of (Y) is a (List)) *then*
  Assert(the Representation of (Y) is
    (Finitely Enumerable))

As a consequence, a new statement can be made:

(4) The Representation of (Y) is
    (Finitely Enumerable)

We now examine the following transformation:

provided (There exists a (Z) such that
  ((X) is a (Z)) and ((Y) is a (Set of (Z)s)
  and (the Representation of (Y) is
  (Finitely Enumerable)))):

```
(X∈Y) => begin
         for each a such that a∈Y do
           if X=a then Return(true);
         Return(false)
         end
```

The enabling condition of this transformation is satisfied by assertions (1),(2), and (4) above. Hence,(X∈Y) can be rewritten as:

```
(5) begin
    for each a such that a∈Y do
      if X=a then Return(true);
    Return(false)
    end
```

Now the following transformation applies:

provided the Representation of (Y) is a (List):

for each a such that a∈Y do S(a)

```
=> begin list t; t ← Y;
     while t ≠ NIL do
       begin
       S(head(t));
       t ← tail(t)
       end;
   end
```

The latter transformation maps the text of (5) into a concrete implementation of the membership predicate X∈Y when Y is a list.

If the representation for Y had been an array instead of a list, the underlying concrete code generated would have been different. For example, consider the following transformation:

provided the Representation of (Y) is
  an (Array [M:N]):

for each a such that a∈Y do S(a) =>

```
  begin integer i;
    for i←M step 1 until N do S(Y[i])
  end
```

Applying this transformation to (5) above, in the event Y is represented by a linear array, maps (5) into the following underlying concrete program:

```
begin integer i;
for i ← M step 1 until N do
  if X=Y[i] then Return(true);
Return(false)
end
```

9. Conclusions

By means of a series of brief examples, we have attempted to demonstrate how programs could be both improved and refined using source-to-source program transformations and program manipulating algorithms.

A much more extensive set of transformations is required to achieve effective coverage of the range of useful manipulations that tend to occur in practice. Many groups of these have been collected in the *Irvine Program Transformation Catalogue* [22], which provides, among other things, a more extensive and precise treatment of many of the topics touched upon only lightly in this brief exposition.

We have omitted here any considerations of how a programmer using an interactive program manipulation system would specify the particular transformation he wishes to apply, and the locus within the program where he wishes to apply it. These matters are treated in a document explaining our program manipulation system *Arcturus*[26].

Despite the limitations of our discussion, we believe that the examples examined give a glimpse of the potential utility of program manipulation techniques for improving and refining programs.

References

[1] Wirth,N., Program Development by Stepwise Refinement, *CACM 14,4* (April 1971),221-227.

[2] Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R., *Structured Programming*, Academic Press, London, (1972).

[3] Shaefer, M., *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, Englewood Cliffs, N.J., (1973).

[4] Allen, F.E., and Cocke,J., A Catalogue of Optimizing Transformations, in *Design and Optimization of Compilers*, (R. Rustin ed.), Prentice-Hall, Englewood Cliffs, N.J., (1972), pp. 1-30.

[5] Aho,A.V., and Ullman, J.D., *The Theory of Parsing, Translation, and Compiling, Vol. II*, Prentice-Hall, Englewood Cliffs, N.J.,(1973).

[6] Allen, J.C., and Luckham, D.C., An Interactive Theorem Proving Program, *Machine Intelligence 5*, (Meltzer, B., and Michie, D. eds), American Elsevier, New York, (1970), 321-336.

[7] Cheatham, T.E., and Wegbreit, B., A Laboratory for the Study of Automatic Programming, *Proc. AFIPS Spring Joint Computer Conference, Vol. 40*, AFIPS Press, Montvale, N.J., (1972), 11-21.

[8] London, R.L., Proving Programs Correct: Some Techniques and Examples, *BIT 10* (1970), 168-182.

[9] Waldinger,R.J., and Levitt, K.N., Reasoning about Programs, Tech. note 86, A.I. Center, Stanford Research Institute, Menlo Park, Cal., (October 1973).

[10] Manna, Z., and Waldinger, R.J., Toward Automatic Program Synthesis, *CACM 14,3* (March 1971), 151-165.

[11] Green,C., Application of Theorem Proving to Problem Solving, *Proc. IJCAI 1*, Washington, D.C., (1969), 219-239.

[12] Lee, R.C.T., Chang, C.L., and Waldinger,R.J., An Improved Program Synthesizing Algorithm and Its Correctness, *CACM 17,4* (April 1974), 211-217.

[13] Low, J.R., Automatic Coding: Choice of Data Structures, doctoral dissertation, STAN-CS-74-452, Computer Science Department, Stanford University, Stanford, Cal., (1974).

[14] Schwartz, J.T., Automatic Data Structure Choice in a Language of Very High Level, *CACM 18,12* (Dec. 1975), 722-728.

[15] Hewitt, C., and Smith, B., Towards a Programming Apprentice, Working Paper 90, A.I. Lab., MIT, Cambridge, Mass. (Jan. 1975).

[16] Knuth, D.E., Structured Programming with *goto* Statements, *C. Surveys 6,4* (Dec. 1974),261-301.

[17] Ingalls, D., The Execution Time Profile as a Programming Tool, in *Design and Optimization of Compilers*, (Rustin, R. ed.), Prentice-Hall, Englewood Cliffs, N.J., (1972), pp. 107-128.

[18] Knuth, D.E., An Empirical Study of FORTRAN Programs, *Software -- Practice and Experience 1* (1971), 105-133.

[19] Moses, J., Algebraic Simplification: A Guide for the Perplexed, *CACM 14,8* (Aug. 1971), 527-537.

[20] Beckman, L., Haraldson, A., Oskarsson, O., and Sandewall, E., A Partial Evaluator, and Its Use as a Programming Tool, DLU 74/34, DatalogiLaboratoriet, Inst. f. Informationsbehandling, Uppsala University, Uppsala, Sweden,(Nov. 1974).

[21] Loveman, D.B., Program Improvement by Source-to-Source Transformation, *Proc. 3rd ACM Symposium on Principles of Programming Languages*, SIGACT-SIGPLAN, ACM, New York, (Jan. 1976), (to appear *JACM*).

[22] Standish,T., Harriman, D., Kibler, D., and Neighbors, J., The Irvine Program Transformation Catalogue, Computer Science Department, U.C. Irvine, Irvine, Cal., (Jan. 1976).

[23] Reynolds, J.C., Transformational Systems and the Algebraic Structure of Atomic Formulas, *Machine Intelligence 5*, (1970), 135-151.

[24] Plotkin, G.D., A Note on Inductive Generalization, *Machine Intelligence 5*, (1970),153-163.

[25] Standish, T., Kibler, D., Neighbors, J., and Winkler, T., Verifying Source-to-Source Transformations, (in preparation).

[26] Standish, T.A., *Arcturus*: An Interactive Program Manipulation System, (in preparation).