Title
Research on interactive program manipulation : final report

Authors
Smith, David A.
Standish, Thomas A.

Publication Date
1979

Peer reviewed

RESEARCH ON INTERACTIVE PROGRAM MANIPULATION

FINAL REPORT

DAVID A. SMITH AND THOMAS A. STANDISH

SUPPORTED BY THE NATIONAL SCIENCE FOUNDATION
UNDER GRANT MCS75-13875 A01

TECHNICAL REPORT 146

DEPARTMENT OF INFORMATION AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA, IRVINE
IRVINE, CALIFORNIA 92717

DECEMBER 1979

Abstract

This Final Report of research conducted under the support of NSF grant MCS75 13875 A01 summarizes results and accomplishments of our research in interactive program manipulation.

Briefly, we developed the Irvine Program Transformation Catalogue which organizes, names, and classifies over 100 groups of program transformations. We discovered how to get a computer to chain together large numbers of simple program transformations with minimal interactive guidance, minimal wasted search, and high efficiency, resulting in dramatic net program improvements. We validated our chaining concepts by building the SPECIALIST system, reported on in the doctoral dissertation of Dennis F. Kibler. We discovered how to verify that transformations in our system preserve program equivalence. Finally, we developed stepwise refinement transformation techniques for mapping abstract program forms into concrete program forms involving the use of exchanges between statements in languages of assertion and relation (i.e. logic) and languages of imperative intent. The refinement transformations are mediated by drawing on knowledge and exchange laws from "micro-worlds", such as the "set", "array", and "sequence" microworlds.

TABLE OF CONTENTS

Table of Contents, cont.

1. Introduction

We develop computer programs to satisfy many goals simultaneously. For example, we desire our programs to be efficient, correct, comprehensible, and amenable to change and improvement.

Unfortunately, these goals conflict. In order to attain efficiency, we must often supply descriptions of representational detail that make programs less concise and legible. On the other hand, lucid, concise, detail-free programs may be easy to understand but may lack required efficiency.

Thus, an effective program develpment methodology must come to grips with the fact that we require different program representations for different purposes -- the principal clash being that we need clear, concise, detail-free programs for human consumption, but that for mechanical efficiency, we need highly detailed, concrete program representations.

If we acknowledge the necessity of having both human-oriented and machine-oriented program forms, then it becomes desirable to have a means for converting between them mechanically -- one that is easy, efficient, and reliable. Assemblers, compilers, and macro-preprocessors are examples of transformers that perform reliable, efficent conversion between program forms that are easier for people to use into program forms better suited for mechanical execution.

If we acknowledge the impracticality of using manual means to perform conversions between these program forms, we are led to seek machine-based methods. Thus, the quest for mechanical program transformation systems is essential to the evolution of effective, practical program development methodologies. Where possible, we would like to accomplish the required transformations by purely automatic means, and where this proves infeasible, we would like to accomplish it with a practical amount of interactive human guidance.

Our research has been aimed at identifying and solving the problems that must be solved to produce practical program development systems based on advanced transformational techniques.

This final report summarizes our progress and accomplishments.

1.1 Summary of Accomplishments

Collecting, Naming, and Classifying

One of our first research tasks was to take stock of known program transformations. We identified two good ways of expressing program transformations: (1) writing program manipulating procedures, and (2) writing pattern-directed exchange laws. We found certain kinds of program transformations that people use which we could not express as computer programs. For these, all we could do was to give examples and to state general principles. Our efforts to collect, name, and classify program transformations resulted in the publication of the Irvine Program Transformation Catalogue [Standish 1976], which contains over 100 groups of program transformations.

Automatic Chaining

If we are to build practical program transformation systems, we must avoid putting the user in the decision-making loop at too fine a grain of detail lest he become overwhelmed with trivia. Many program improvement transformations result from application of large numbers of simple, low-level transformations, such as (if true then A else B => A) which are analogous to simplification rules in algebra such as (X + 0 => X). It becomes important not to force the user to specify the application of each such low-level transformation. To build practical systems, therefore, it becomes necessary to find ways to chain the application of many low-level simple transformations to provide bigger, global transformational effects, and to provide the user directing the activity of the system interactively with a range of granularities of application of its constituent tools.

Loveman [1977] has noted that when certain transformations are applied, certain others immediately suggest themselves. For instance, if certain input variables to a procedure are known to have constant values, we may substitute the constants for the variables (up to the point where the variables are reassigned new values). This may enable arithmetic to be performed, resulting in still other constants, which may, in turn, be propagated. The propagation may lead to useless assignments and to dead variables, which may be eliminated. Logical conditions may evaluate to true or false, arms of conditional expressions may drop, loops may collapse, and so on.

The problem of automatic chaining is challenging because the search space for constructing chains is "doubly bushy" -- not only must we select one of many places in the program to transform, we must also select one of many possible transformations to apply at each possible place.

Problems of excessive back-up, search catastrophe, and inefficiency must be solved.

We built the SPECIALIST program transformation system (cf. [Kibler 1978]) to demonstrate our solution to this problem. We were able to achieve dramatic program simplifications with 70 to 90 applications of program transformations, no back-up, and either no user guidance or very slight user guidance.

Validity

In order to build practical program transformation systems, we must be able to convince users that their transformed programs have the same net effect as their original programs. Thus, it is important to be able to prove that constituent transformations in a system preserve program equivalence.

We devised proof techniques relying on manipulating expressions in Dijkstra's weakest precondition formalism, to prove transformations valid [Kibler 1978].

Refinement Transformations

In order to be able to move from high-level, abstract programs to concrete, efficient implementations, we must understand how to transform abstract programs into concrete refinements. Sometimes these refinements involve supplying a data representation and organizing the details of the processing of that representation to imitate the actions specified at the higher level of abstraction.

The most ambitious form of program refinement is to take input and output assertions of what the program is to do, and to generate suitable auxiliary data structures and a program which accomplishes the desired function.

The definition of language extensions in extensible languages provides another kind of refinements wherein, when we write a program in the extension, we are shown how it maps down into the base language into an equivalent concrete underlying form. The technology for extensible language definitions of the new in terms of the known is here now, and it works. When we program in extensions, we can write programs in lucid, concise, detail-free form, and we can rely on the definitional mappings to transform these human-oriented forms into base language.

In yet another form of program refinement the data entities are preserved by the refinement, but attributes or properties are added which permit the use of new operators. One reason for doing this is to introduce operations with

which the algorithm can be performed more efficiently. In a sense this step is always necessary when providing an implementation for operators not directly supported by the machine -- the added predicates are in fact the conventions for implementing the high level constructions in the lower level machine.

We have studied examples of these modes of program refinement using a mixture of static, assertion language (the language of logic) and dynamic, imperative language (the language of programming). We have found these techniques to be quite powerful for exhibiting a program clearly and for capturing and parititioning essential aspects of the design.

1.2 Prospects for Applications of Our Results

We think sufficient progress has been made in developing basic understanding of program transformations that the time is ripe for applying the basic knowledge that we and others have discovered. In our view, it is time to attempt designing, building, and validating production engineered, program transformation systems.

What is a realistic position to take on what sorts of application systems might now be achievable? What is now within our reach?

To come to grips with what reasonable expectations for such an undertaking might be, let's start by making a few simple observations.

First of all, if we adopt a somewhat loose definition of what a program transformation is, we can observe that the world of computing abounds with program transformations already. Speaking loosely, suppose we say that any process which rewrites a program is a program transformation.

Then we can observe that compilers, assemblers, and macro-preprocessors fall under the scope of program transformations.

In program optimizers, transformations that improve program performance are in common use. For example, redundant computation can be eliminated by optimizing transformations such as removing invariants from loops, eliminating common subexpressions, unrolling loops, eliminating assignments to dead variables, and so on.

New levels of abstraction in extensible languages are often defined by showing how to replace the new with the known. John Backus' FFP language [Backus 1978], of which several implementations exist, is an example of the use of this principle.

Program transformations have also been used to give clear definitions of programming language semantics as in the definition of for list elements in for statements in the Algol 60 Report.

Realistically speaking then, if we are willing to adopt a broad definition of what a program transformation is, then we can observe that many different sorts of program transformations are in common, practical use today -- they are here, they work, they help, and they are used.

One model of how the new results on program transformations will be applied is to envisage that they will be added in an evolutionary fashion to programming environments already in use. That is, application will come

by incremental absorption, and by trial and error. This has happened in the past and will likely continue to happen. What might we realistically expect as far as these gradual improvements are concerned?

In our opinion, one important existence proof to look at is the Harvard Program Development System (PDS) [Cheatham 1979], currently operational in prototype form for friendly users. A production engineered version could very likely be built on the basis of present experience.

The Harvard PDS supports a program development and maintenance methodology based on transformations. The view is that a user (or group of users) defines an extension of the base language (EL1) to provide the data, notation, and operations well-adapted to a particular application, and one programs and maintains concise, lucid, detail-free modules in the extension. Definitional transformations map extension modules into the base language. The Harvard PDS supports version control disciplines that enable incremental rederivation of modeules affected by incremental changes, and it uses refinement transformations to support a software development methodology based on stepwise refinement and the use of abstractions. This permits programmers, and more importantly, maintainers to use concise, lucid program forms for ease of comprehension and change, while permitting efficient, less lucid, concrete realizations to be derived.

While the Harvard PDS contains a broad collection of tools such as editing tools, analysis tools, and execution tools -- it depends on transformation tools to support its view of program development and maintenance.

Let us define a <u>programming environment</u> as that collection of computer-based tools that supports program development and maintenance. Typically a programming environment has a data base, which is the repository for programs, test data, documentation, production schedules, and other intermediate entities under development in a programming project. It also has a collection of tools for the creation and management of text, the execution and debugging of programs, the management of files, and for other programming project activities.

The place for program transformations, in our view, is as part of the tool kit in a programming environment. Program improvement transformations clearly belong in such a tool kit, and we can take incremental steps to expand the set currently in use in conventional optimizing compilers to include some of the more ambitious transformations that we and others have recently discovered. It would be feasible in today's state of the art to implement all the transformations in the <u>Irvine Program Transformation Catalogue</u> in such an environment. Additionally, we could begin to add sophisticated transformational packages, such

as those in our SPECIALIST system, which chain low-level transformations together semi-automatically to provide dramatic performance improvements. Furthermore, as the Harvard PDS shows, we can construct environments based on the use of stepwise refinement transformations and abstract program forms. This much we can certainly do in today's state-of-the-art. The utility and appeal of such systems in production environments needs to be validated, but initial results reported by Squires [1979] at the San Diego Ada Environment Workshop indicate the approach is already highly successful for one advanced technology programming shop.

Beyond this, what should one hope for?

It seems to us that the key problem is one of integration. We need to be able to combine the capabilities of the contributions of many basic computer science researchers into a common system in which their modules can cooperate to provide transformational power.

In our own research, we made a serious attempt to use the results of previous investigators by importing Hearn's REDUCE system from the University of Utah [Hearn 1971], and by using it as the base for implementing our program transformations. By this means, we hoped to utilize Hearn's work on algebraic and logical formula manipulation to provide us with a prefabricated subsystem for assisting with subproblems in formula simplification which arise in program transformation work.

We were successful in bringing up REDUCE on our own machine and we were successful in implementing a small package of our own program transformations on top of it. But the system got too large for efficient execution on our machine, and it became too expensive for our computing budget, so we had to abandon the approach and build a more modest stand-alone system from scratch, simultaneously abandoning hope of using the work of others to improve the power and effectiveness of our system.

We would have liked to have been able to import even more capabilities and to have been able to absorb them in our system. For example, we had a need to be able to propagate logical assertions (such as (N>1)&(N<K)) throughout a section of a program and to deduce facts affecting program simplification (such as simplifying if N<0 then A else B to B with respect to such a propagated predicate). It would have been useful to be able to import a good logical deduction system, such as Luckhams's or Boyer and Moore's to deduce that (N>1)&(N<K)&(N<0) ==> false. Again because of economic constraints we were unable to take this step, and thus we were unable to realize the transformational power that coupling of such systems with our own could have implied.

This raises the issue of how investigators in our subfield might expect to couple the separate capabilities of system modules produced by separate investigators in the future within reasonable economic constraints. We think this issue of integration is the key issue to be solved in building practical, effective, sophisticated transformation systems of the future.

Fortunately, there is good reason to be sanguine that the problem of integration can be solved. The day is fast approaching when we can have PDP-10 size machines on chips with million word memories on our desktops for modest expense. Current technology is nearly at this point. When this happens, we can have million word computer programs under development, executable by dedicated processors for very reasonable expense. And such research will be within the easy reach of NSF computing research budgets that may run between five and ten thousand dollars.

We believe the time is nearly at hand when experiments at integration and validation should be attempted. We further conclude that such experimentation shoud take place in the framework of total programming environments -- integrated, complete tool sets for program development and maintenance -- and that the next hot topic in program transformation research is developing the understanding to build effective programming environments as platforms for later attempts at absorption and integration of advanced transformational techniques.

2. Overview of Our Program Transformation Approach

Our research has been aimed at developing techniques in program automation enabling realization of very high level programming languages -- languages which permit problem solutions to be expressed using specialized notation and concepts drawn from a particular scientific discipline or application area.

A significant focus is the development of refinement transformations that can accept concise, high-level program specifications which manipulate abstract data, and can produce from them low-level programs which operate efficiently on underlying data representations. We have aimed further to learn how to employ such transformations in new high-level interactive programming media, in which the user writes concise, high-level programs and then directs the production of low-level mechanical detail through the application of selected program transformations. The high-level programs are to be specified in a language free from commitment to underlying data representations, allowing the widest possible selection of implementation detail and permitting exchange of underlying data representations where program performance measures reveal it to be advisable.

We have also dealt with program improvement transformations which operate within a given level of language. Such transformations are used to transform a clear, concise statement of a program into an equivalent version which is more efficient but perhaps less legible and less comprehensible.

Rather than regarding improvement and refinement of programs as something performed behind the scenes, to be done, say, in a machine language which the programmer never sees, we rely on source to source transformations explicitly visible to the programmer.

We envisage that a programmer should write a beautifully structured program, at some more or less detail-free level, and should then apply transformations to map it into lower levels of representation, or to improve its performance and make it more efficient. The mechanical assistance we provide in producing low-level details, enables more reliable production of low-level representations, and permits programs to be written and understood at high, concise levels. Also, the cost of changing underlying representations is greatly reduced by reliance on mechanical means.

We are convinced that, in the long run, improvements in programmer productivity will rest upon making the machine a more talented partner in the process of program production. An important ingredient in achieving this overall goal is to find ways of programming the computer to fill in most

low-level programming details by itself and to transact with much less detailed process and data specifications than it does at present. Thus, our efforts have been aimed at finding practical, effective, source to source transformation systems that give us new leverage in the tasks of mechanical program production.

2.1 Historical Background and Perspective

The idea of adding source to source transformations to our programming systems seems to be enjoying a new round of interest. Knuth has become an advocate of this idea, and his paper "Structured Programming with go to statements" [Knuth 1974] seems to have done much to rekindle interest in the subject. And yet, transformations aren't exactly a new idea in programming. In this section we present a bit of the history of the source to source transformation idea, and we place our work in perspective.

Production Systems

One of the most frequently used ways of organizing source to source transformations is as "pattern-directed" transformations. By this is usually meant a collection of "productions" applied in some prescribed order, wherein each production works by pattern matching, extraction, and substitution.

More specifically, a production of the form P -> E has a pattern P and a right hand side E. We first attempt to match P against some source program fragment S, and if P matches S, we develop a correspondence between variables in P and matching subexpressions of S. We might represent this correspondence, for example, as a set of ordered pairs (Vi,Si) where Vi matches Si for some set of pattern variables Vi ($1 \leq i \leq n$), and some set of subexpressions Si of S. In the special case that the set of variables Vi is vacuous, the pattern P must be identically equal to S. The (possibly vacuous) correspondence (Vi,Si) is then used in an act of substitution on the right hand side E. Substitution consists of simultaneous replacement of occurrences of the Vi in E with corresponding Si. The substituted version of E replaces S, and constitutes the transformed (or derived) version of S. Sets of productions can be organized into programs that repeatedly transform an initial expression and derive a final result. This basic idea appears in a remarkably broad number of forms, as shown in the remainder of this section.

Markov Algorithms

One of the most basic forms of production system is the Markov Algorithm in which the productions P -> E operate on strings S. In this case, it can be shown that Markov Algorithms can compute any computable function [Markov 1954]. Thus, production systems are a provably powerful basis for computation.

Programming Languages Based on Pattern-Directed Processing

The basic notion of pattern matching followed by substitution has been used in many programming languages as a basis for the description of algorithms, and as a basis for portions of the language processor [Sammet 1969]. The interesting fact is that the data structures on which pattern-directed processing operates can vary from strings, to graphs, to lists, to formulas, even while the surface notation relies on a more or less common form of "production". For example, COMIT and SNOBOL are string processing languages using productions; FLIP is an extension of LISP that manipulates list structures by pattern-directed manipulation; Formula Algol and FAMOUS are formula manipulation languages that use productions to manipulate algebraic formulas; and Floyd-Evans productions are a system of pattern-directed transformations used to parse sentences in context-free languages [Aho 1973]. AMBIT-G is a programming language that manipulates graphs by pattern-directed transformations [Christensen 1965].

In several of these languages, productions can be labelled, and after a successful application of a production, a transfer of control to another designated labelled production can be specified.

Transformations in Compiler Optimization Theory

Source to source transformations play a role in the theory of program optimization [Aho 1973]. For example, Aho and Ullman show how to use four basic transformations on straight line code to eliminate useless assignments and redundant computations and to find an optimal program under certain cost functions. Transformations can also be used to unroll loops, to optimize Boolean expressions, and to simplify programs by, for example, doing arithmetic at compile time, replacing subroutine calls with partially evaluated open subroutine texts, and replacing exponentiation to a small integer constant with multiplications (though in these latter cases, it can be argued that pure forms of source to source transformations are not the natural formalism to use).

Macros and Syntax Macros

One of the most successful uses of source to source transformations is in the definition and call of assembly language macros [McIlroy 1960]. In this case, it is interesting to note that adequately versatile macro facilities in assembly languages normally include a variety of useful "bells and whistles" permitting conditional and iterative assembly, created symbols, and nested definitions and calls.

Syntax macros operate at a level above that of assembly language, but with much the same philosophy as assembly language macros [Leavenworth 1966]. The significant difference is that the calling forms of expressions defined with syntax macros can act as if they were grammatical extensions of high-level languages defined using context-free grammars.

Transformations Used in Language Definitions and Extensible Languages

It is interesting to note that transformations are used with some popularity to define the semantics of various source language constructions. For instance, in the Algol 60 Report [Naur 1963] for statements are defined using source to source transformations. For example, in section 4.6.4.2, the semantics of the "step until element" is given by showing that a statement of the form:

```
for V := A step B until C do S
```

would be equivalent in meaning to:

```
     V := A;                                    (1)
L1:  if (V-C)*sign(B)>0 then
         goto element exhausted;
     statement S;
     V := V + B;
     goto L1;
```

Another example of such a definition is in the explanation of the meaning of a while-do statement, wherein while B do S is defined by showing how to implement its meaning in terms of a transformation to the equivalent lower level construction:

```
L1: if B then
        begin statement S; goto L1 end
```

Source to source transformations have also been used to define the meaning of Backus' FFP Languages [Backus 1978]. Here, the meaning of a top level construction is given by a defined series of self-replacing reductions into lower level constructions. Similarly, Teitelman's extension of INTERLISP, called C-Lisp, defines high-level Algol-like constructions on top of LISP by a technique of reduction into LISP. The same technique is used in defining extensible languages as in Bell's PROTEUS [Christensen 1969], an extensible language with extensions based on the idea of transformations, and in the Galler-Perlis extension capability in ALGOL D [Galler 1967].

Thus, we see that the idea of pattern-directed, source to source transformations has received widespread attention and has been explored in a rather remarkable variety of circumstances. In the next section we consider the issues that arise in making these techniques completely formal.

2.2 Implications of Simple Source to Source Transformations

Oftentimes, the use of source to source transformations in the literature hides a number of underlying problems. The dirt is swept under the rug, so to speak. For example, in the use of transformations to define "for list" elements in the Algol 60 Report, cited above, consider the label L1. Such a label must be generated carefully for each application of the reductive transformation that maps "for V := A step B until C do S" into an underlying equivalent loop. In particular, L1 must not conflict with the use of names local to its lexical scope of substitution -- that is, it cannot be identical to the names of identifiers, functions, labels, or reserved words used locally, and it cannot be the same for nested applications of the transformation. If it were, then nested for statements would have conflicting labels in the same block. This means that we must face two problems immediately:

(a) The problem of generation of unique names, and
(b) The problem of naming conflicts.

Consideration of the second of these problems leads to the conclusion that we must be able to enumerate the names used locally in the lexicographic scope where a transformation is to occur, and we must be able to use this enumeration to control generation of distinct, non-conflicting names. Other situations occur where we must be able to change names systematically so as to be able to avoid naming conflicts. This occurs, for instance, in substituting a partially evaluated text of a procedure in place of a procedure call -- names normally thought of as non-conflicting, because of their definition in disjoint blocks can suddenly be brought into conflict through such a substitution. We are required to be able to examine any point of a program text to discover the local names in effect.

Considering further the example from the Algol 60 Report, we see the expression "goto element exhausted". The Algol 60 Report describes the meaning of this expression as follows:

> "... element exhausted points to the evaluation according to the next element of the for-list, or if the step-until-element is the last of the list, to the next statement in the program".

In other words, the binding of the label element exhausted depends on whether the "step until element" is the last in the "for list" or not. To handle such a situation, conditional generation of program text is needed at the very least. In actual fact, to handle the transformation of a moderately complex Algol 60 for statement (involving, say, nesting and a "for list" with variety), much more mechanism is needed than the Algol Report gives explicitly. For

example, consider transforming the following <u>for</u> statement (cf. Algol Report 4.6.2) into underlying (<u>for</u> statement free) text:

Example (A)

```
for k := 1, V1*2 while V1<N do
    for j := I+G, L, 1 step 1 until N, C+D do
        A[k,j] := B[k,j]
```

We use transformation (1) for the "step until element", together with the following transformation for the "while element":

```
for V := E while F do S

    =>

L3: V := E;
    if (not F) then goto element exhausted;
    S;
    goto L3;
```

If we follow the Algol Report literally, we must duplicate the statement S in the body of each <u>for</u> statement, once for each "for list" element. Abstractly, this would mean that for a <u>for</u> statement of the form:

```
for V := V1, V2, ..., Vn do S
```

with a "for list" V1, V2, ..., Vn we would have to produce an underlying transformed text equivalent to that resulting from transforming the statement sequence:

```
for V := V1 do S;
for V := V2 do S;
    . . .
for V := Vn do S;
```

Transforming each statement in this latter sequence produces a copy of the transformed version of S. If S is a large piece of text this is needlessly wasteful. Thus, the remedy adopted in many compilers is to make S into a closed subroutine, and to call it from several different places. Just as in manual programming, this is a circumstance where the same piece of text needs to be executed from a number of different points of call -- the exact situation where programmers create closed subroutines. Hence, a source to source transformation facility needs to be able to create and name closed subroutines to handle the <u>for</u> statement transformation in a desirable fashion.

Taking Example (A) above, here is what the output of such a transformation might look like:

```
    k := 1;
    call P1;
L3: k := V1*2;
    if (not (V1<N)) then goto E1;
    call P1;
    goto L3;
E1: goto A1;
        procedure P1
            begin
            j := I + G;
            call P2;
            j := L;
            call P2;
            j := 1
        L1: if (j-N)*sign(1)>0 then goto E2;
            call P2;
            j := j + 1;
            goto L1;
        E2: j := C + D;
            call P2;
            goto A2;
                procedure P2
                    begin
                    A[k,j] := B[k,j];
                    end P2;
        A2: Empty Statement
            end P1;
A1: Empty Statement
```

This text corresponds rather well with the kind of code produced by typical non-optimizing compilers. It is clearly better to produce the closed subroutines for bodies of <u>for</u> statements, not only because this avoids duplication of large bodies of code (as would have happened had we reproduced the body of P1 twice in the above example, as straightforward application of the Algol Report transformations would have dictated), but also because, in a recursive language such as Algol 60, the use of the procedure call mechanism handles recursive calls from within the bodies of <u>for</u> statements automatically.

Yet, the transformed text above fails to meet Algol 60's requirements, since all procedures must be declared in the heads of blocks. Thus, to produce legal Algol 60 code, we must move the procedure declarations upward through the surrounding body of program text from the point of application of the transformation to the head of the nearest enclosing block.

Looking more closely at the above text, we see instances where program simplification would help. For example, the line labelled L1 could be simplified by using transformations such as:

```
sign(1) => 1
(j-N)*1 => (j-N)
(j-N)>0 => j>N
```

resulting in the net transformation:

```
L1: if (j-N)*sign(1)>0 then goto E2

      =>

L1: if j>N then goto E2
```

Furthermore, several goto "chains" can be simplified, as in transforming:

```
    goto E1;
    . . .
E1: goto A1;

      =>

    goto A1;
```

This brings up an important point -- while programmers would rarely create programs containing unsimplified expressions such as (j-N)*sign(1)>0, or containing goto "chains", it is readily apparent that unsophisticated source to source transformations produce exactly these things. Thus, the need arises for program simplification devices to clean up the preliminary outputs of transformations.

We see that the consideration of one of the examples of program transformation in the Algol 60 Report has led us to the consideration of even more problems. Not only must we be able to create names and resolve naming conflicts (as in problems (a) and (b) above), we must also be able to solve problems such as the following:

(c) The problem of moving text (such as procedure declarations) to new locations in the text surrounding the point of application of a transformation, and of making required associated changes in naming and control.

(d) The problem of simplification of program text (at least to eliminate some of the more flagrantly wasteful results of the application of source to source transformations).

The authors of the Algol 60 Report wisely avoided the "can of worms" that would have resulted in their making the details of program transformations explicit. While their use of English text to describe the semantics of Algol was not without its pitfalls [Knuth 1967], any attempt to define

reductive source to source transformations from well-formed Algol 60 into a subset of well-formed Algol 60 would have run into problems that would likely have caused the text of the Algol Report to swell, thereby diminishing its appeal.

Several other problems need to be considered in mechanizing source to source transformations:

(e) The problem of error and diagnostic messages.
(f) The problem of efficiency.
(g) The problems of generality and convenience.

Error and Diagnostic Messages

Suppose that we map a program written at a high, abstract level into an underlying text that relies on some low-level underlying representation. What happens when we encounter an error in the operation of the underlying representation, such as attempting to divide by zero? The user of a program wants to understand the nature of an error in problem domain terms. Attempting to divide by zero at an underlying representational level might imply attempting to take the area of a triangle with one side longer than the sum of the other two, or taking an average score of an empty set of scores, up at the problem domain level. The user wants an error (or diagnostic) message that identifies the nature of the trouble in problem domain terms. It is useless to him to have the program report an error in a underlying representation of which he is not even supposed to be aware, such as "illegal argument to system function", or "push down list overflow". Source to source transformations are a "one way street" in the sense that higher-level meanings are reduced to combinations of lower level meanings. The translation of underlying error conditions back up into problem domain terms goes in the opposite direction to that of a purely reductive transformation, and requires "anti-reductive" features of the transformation.

Efficiency of Transformations

McIlroy once commented on source to source transformations as follows [McIlroy 1969] (p 51):

> Another difficulty is that, to date, most of the implementation-independent schemes that have been reduced to practice yield appallingly slow compilers.

It is possible for source to source transformations to reduce their initial inputs to final outputs in a number of "passes" or iterations. This happens especially when a given transformation works by manufacturing calls on other

subsequent transformations and embeds them in its output. The output must then be scanned to detect new calls and to transform them, in turn. It is appealing to write transformations that work this way, since it allows the solution of problems by breaking them down into simpler sub-problems. This tends to add to the clarity of the structure of the overall solution. Yet, the price to be paid for this structuring is repeated rescanning of source text (involving perhaps parsing or tree-walking, depending on what internal form is used to represent the text). Here is an obvious case where good structuring might conflict with efficiency. One rather well known case where this phenomenon occurs is in macro assemblers that rely on deeply nested macro expansions to produce the final code. Such macro programs may be pretty, but they can often take an unexpectedly long time to assemble.

While avoidance of inefficient internal representations and costly reparsing may help alleviate this problem, it seems inherent that the source to source transformation approach contains this potential avenue of abuse by users.

We have explored techniques to limit the searching which must be done in order to decide which transformations ought to be attempted. This is described later in this report in conjunction with our SPECIALIST system.

Problems of Generality and Convenience

As we have seen, simple source to source transformations, based, say, on a macro-like style, or alternatively, on a pattern-directed transformation style, using production systems that act like labelled Markov Algorithms, are encumbered with a number of problems surrounding their effective use in the context of program manipulation. The solution of these problems fosters the creation of special mechanisms to deal with names and name conflicts, to simplify programs, to pass error messages back up to the untransformed source level, and to promote efficiency. While these mechanisms are supposedly not required for generality, in as much as simple Markov Algorithms can do anything that is computable, they are required for convenience in order to avoid what Perlis calls the problem of the "Turing Tar Pit" -- that world where everything is possible but nothing of interest is easy. Practical considerations require us to provide mechanisms that can be used with ease to build effective transformations of interest without unacceptable volumes of programming, or unacceptable indirectness of representation.

As our aspirations grow with regard to the kinds of transformations we would like to be able to write easily, the mechanisms required to support them grow accordingly.

As we tackle the problems of system building to support more ambitious transformations, such as recursion removal, sophisticated program simplification, change of representations, and so forth, the simple macro and Markovian models grow less and less adequate. While simple Markovian Production Systems are an important special case, allowing us to write simple transformations in simple notation, more is needed beyond the pattern matching, extraction, and substitution paradigm to accomplish more sophisticated aims with ease, as we have demonstrated above.

3. The Irvine Program Transformation Catalogue

The Irvine Program Transformation Catalogue [Standish 1976], to quote its title page, provides "A stock of ideas for improving programs using source to source transformations". The transformations in the Catalogue cover a wide range of power and sophistication, and are documented so as to be easily and precisely understood. Consideration has been given to the conditions under which the transformations can be applied while preserving program equivalence, and while most often the "forward" direction of a transformation is designed to yield a faster running program, the inverses of many transformations are also valid from the point of view of correctness. In some cases different names are applied to the same transformation when it is applied in the forward or the reverse direction. For example, the process of "Procedural Abstraction" (p 70) refers to identifying common code patterns which can be replaced by calls to a parameterized subroutine. This is precisely the inverse of the transformation of "Eliminating Calls" (pp 44-53).

Many of the transformations described in the Catalogue are expressed formally as rules of exchange. In this form a transformation is described as two program fragments, written in terms of pattern variables and constants, which are asserted to be equivalent to each other (given the satisfaction of appropriate enabling conditions). As explained before, a transformation in this form is written as P => E, where P is a pattern and E is the right hand side. We first attempt to match P against some source program S, and if P matches part of S, we develop a correspondence between variables in P and corresponding subexpressions of S. The portion of S matched by P is replaced by a substituted version of E. Each pattern variable occurring in E is replaced with the value bound to the pattern variable during matching. The following are examples of such transformations:

```
x*(y+z)                      =>  x*y + x*z

b and (b or c)               =>  b

x + (if b then y else z)     =>  if b then x+y else x+z
```

Some of the more powerful transformations in the Catalogue cannot be described conveniently in terms of exchange rules. In many cases, however, program manipulating procedures are given, which provide a sufficiently precise definition. For example, consider the transformation called "Useless Assignment Elimination" (pp 13-15):

Repeatedly delete assignments of the form x := E,

where x is not used subsequently in the program and where E is side-effect free.

This transformation is made more precise with the following algorithm:

> Construct a directed graph with a node for each variable. Construct for each assignment an arc from node v1 to v2 whenever v2 := E(...v1...) where E(...v1...) is any expression containing an occurrence of v1. Now put squares around output variables. Retain that portion of the graph connected by at least one path to some square (i.e., to some output variable), and discard nodes not connected to some square by at least one directed path. Now eliminate assignments corresponding to deleted nodes.

Other useful program transformations which cannot be expressed conveniently in a pattern-directed form include expanding procedure calls in line, recursion elimination, elimination of useless declarations, and many others.

Some transformations in the Catalogue rely simultaneously on described (or implied) procedures in addition to pattern defined rules of exchange. An example of this is "Common Subexpression Elimination" (pp 19-20):

> Suppose the subexpression E is used two or more times on the right hand sides of assignments in a straight line sequence of assignments. Let t be a distinct new variable not appearing elsewhere in the program. Provided no variable in E is reassigned in the statements between the assignments to x and y, respectively, and t is a distinct new variable:

```
x := ...E...;
. . .
y := ...E...;

   =>

t := E;
x := ...t...;
. . .
y := ...t...;
```

In the interest of completeness, some of the transformations in the Catalogue are described by means of examples with discussion, for lack of suitable formal methods. It is not the case that such a given example could not be expressed as a program modifying algorithm, as

described above. However, the basic idea in these cases extends beyond the framework of the example given -- and does so in a way that cannot be completely formalized at the present time. One case of this is found on page 64, "Eliminating Search Exhaustion Tests by Data Structure Extension":

```
procedure Search(T,X); array T[1:N]; integer i;
    begin i := N;
    while i>0 do
        if T[i]=X then Return(true) else i:=i-1;
    Return(false)
    end

   =>

procedure Search(T,X); array T[0:N]; integer i;
    begin T[0]:=X; i:=N;
    while (not(T[i]=X)) do i:=i-1;
    Return(not(i=0))
    end
```

This is accompanied by the following discussion:

> This specific program improvement is an example of a general principle, which can be stated as follows. Suppose you are searching to find whether a search space S contains an element X by systematically enumerating elements of S in some prescribed order and comparing them to X. If you extend S to include one more element containing X, which is guaranteed to be enumerated last in the particular order of search used, then there is no need to test explicitly for exhaustion of the search space in your algorithm. Instead, you can generate elements in the extended version of S until finding X, and then you can conclude X was in the original unextended space S, if and only if it was not found in the extension.
>
> Once this general principle has been understood it can be applied not only to tables, but to search spaces of many different shapes and organizations, such as binary trees, list structures, and indexed files.

We do not know how to write a procedure which will test any given program to see if this optimization is applicable. Furthermore, although the generalized transformation in the discussion sounds somewhat like a procedure, the meaning of "extend S" is not at all precise, and may in fact be impossible or undefined.

The purpose of the Catalogue when originally written was to communicate with people. Therefore the choice of presentation of a given transformation was made to promote clarity. Transformations in the Catalogue were presented in three ways: (1) pattern-directed exchange rules, (2) program manipulating algorithms, and (3) examples plus discussion.

In making precise the conditions under which certain transformations can be applied correctly, certain enabling conditions are mentioned, where applicable. Two conditions which are frequently required are commutativity and freedom from side effects. These are defined as follows (p9):

> Let F be a well-formed program fragment (i.e. a phrase in the grammar of the programming language at hand). Let R(F) be the set of variables non-local to F that F either reads but never writes or reads before writing, and let W(F) be the set of non-local variables that F writes (whether or not it also reads them). Here a local variable v in F is a variable used only within F and not elsewhere in the program.
>
> Given two program fragments A and B, we wish to know when it is permissible to exchange their order of execution from A;B to B;A while preserving program equivalence. It is permissible for A and B to read the same read-only variables, but neither A nor B may write into variables the other reads before writing, nor may they write into common variables. In symbols, A and B are commutative provided that
>
> W(A) intersection (W(B) union R(B))
> = emptyset, and
> W(B) intersection (W(A) union R(A))
> = emptyset
>
> . . .
>
> If the program fragment F does not write into any non-local variables (i.e. if W(F)=emptyset), then we say that F is side-effect-free. Some transformations preserve program equivalence only if one or more of the constituent program fragments is side-effect-free. For example, the McCarthy conditional transformation:
>
> a&b => if a then b else false
>
> requires b to be side-effect-free in order to preserve program equivalence, in general.

We believe our Catalogue constitutes a contribution to the state of our knowledge about program transformations, and also serves as a source of ideas which can be used manually in actual programming practice. Although a number of transformations may seem obvious, or applicable only to poorly written programs, it is precisely this class of transformations which must be managed by an automatic programming system so that the programs generated may be considered reasonable by human programming standards. The Catalogue also focuses on non-trivial program improving transformations, and provides a number of ideas for very high level program manipulations.

Among the topics dealt with that concern transformations within a given level of expression are the following: Assignment Forms, Go To Forms and Labels, Conditional Forms Looping Forms, Compound Statements and Blocks, Declarations, Procedures, The Mechanics of the Empty and Undefined Program Forms, and Manipulation of Expressions.

4. The Pattern-Directed Program Transformer, SPECIALIST

The SPECIALIST system [Kibler 1978] is a prototype program transformation system which was built as a test vehicle for our ideas on pattern-directed transformations. Its name derives from its principle use: the "specialization" of general purpose programs to operate more efficiently under restricted circumstances. Used in this manner, the system operates by accepting a program and an initial constraint expressed as a single transformation of the program. SPECIALIST is then able to proceed automatically in identifying and performing additional valid transformations, doing this repeatedly to achieve an improved version of the input program.

The system accepts programs in an Algol-like language, and operates on an internal representation which is equivalent to the syntactic parse tree of the program. This is convenient because the transformations are expressed as patterns which correspond closely to syntactic entities in this kind of language. Naturally this restriction to local transformations places a limit on the kinds of program manipulations that can be performed, but for this price we can achieve much greater efficiency of operation, and can still get quite dramatic and interesting program improvements. In the remainder of this section we discuss the organization and operation of the SPECIALIST system, with examples of its performance.

Examples of Operation

Two examples of SPECIALIST's performance are given here. The total number of productions attempted is given after "rules tried". The total number of productions which were actually applied is given after "successful".

Program #1: Multiplication of matrices A and B

```
for i := 1 to n do
    for j := 1 to n do
        begin
            C[i,j] := 0;
            for k := 1 to n do
                C[i,j] := C[i,j]+A[i,k]*B[k,j];
        end;
```

The constraint is placed on the input program that B is triangular. In the knowledge base, this is expressed as the following transformation:

```
B[i,j]   =>   (if i ≤ j then 0 else B[i,j])
```

The output program is then:

```
for i := 1 to n do
```

```
        for j := 1 to n do
            begin
                C[i,j] := 0;
                for k := j to n do
                    C[i,j] := C[i,j]+A[i,k]*B[k,j];
            end;
```

rules tried: 27
successful: 23
user interactions: 1

Program #2: Determinant of 3 by 3 matrix A

```
begin
    det := A[1,1]*A[2,2]*A[3,3] +
           A[1,2]*A[2,3]*A[3,1] +
           A[1,3]*A[2,1]*A[3,2];
    tmp := A[1,2]*A[2,1]*A[3,3] +
           A[1,3]*A[2,2]*A[3,1] +
           A[1,1]*A[2,3]*A[3,2];
    det := det - tmp;
end;
```

The constraint is placed on the input program that A is triangular. The output program is then:

```
begin
    det := A[1,1]*A[2,2]*A[3,3];
end
```

rules tried: 98
successful: 91
user interactions: 2

In the first example we observe that the inner for loop has been reduced from the range 1..n to the range j..n. This means that the total number of multiplications performed goes from n**3 to n*n*(n-1)/2, which constitutes an improvement slightly better than a factor of two. In the second example it should be noted that manual intervention was necessary at one point to restart SPECIALIST, which had halted with the program in the following state:

```
begin
    det := A[1,1]*A[2,2]*A[3,3];
    tmp := 0;
    det := det - tmp;
end;
```

With an appropriate transformation interactively suggested, SPECIALIST is able to finish this example automatically. It is interesting to note that even before the manual intervention, SPECIALIST has taken a program with twelve multiplies, and produced a program with only two. (In both examples, the "first" interaction consists of the initial

invocation of SPECIALIST.)

System Structure

SPECIALIST is organized as a production system, with the production rules corresponding to steps in the transformation of the program under consideration. Care was taken to avoid computational inefficiency by limiting search operations. A technique was devised for guiding the attention of the system based on the transformations which had recently been performed.

The system consists of the following components: a user interface, a collection of transformations, a small definitional knowledge base, and a task executor. The user interface handles input and output of programs in a form the user can read. The user can also add transformations, change the knowledge base, and modify the chaining among transformations. The transformations are primarily production rules consisting of a pattern to be matched, a substitute pattern, a set of enabling conditions which must be satisfied for the transformation to be applied, and associated directions for possible transformations to try next. The definitional knowledge constitutes some additional transformations which "define" or elaborate properties which may be asserted about the program data. It is by applying one of these assertions to a program that SPECIALIST normally begins its chain of transformations to the final optimized version. The task executor keeps track of where the program is to be examined for applying transformations, and causes applicable transformations to be performed.

Production Rules

The internal representation of programs under SPECIALIST consists of nested trees. This amounts to a LISP-like representation of the parse tree of the program. Pattern matching and the application of transformations take place on subtrees of this internal form of the program. This prevents unintended manipulations such as 3+0*4 => 3*4 which would result from applying the transformation X+0 => X to a simple string representation of the program text.

The rules of the production system are transformations from tree patterns to tree patterns -- or conceptually, from source code to source code. There are no productions which leave temporary "flags" or "markers" in the code, or non-valid program constructions which are later cleaned up by other transformations. Each transformation is constrained to leave the "meaning" of the program invariant. Methods for verifying this property for candidate transformations are described later in this report.

SPECIALIST has transformation rules which are predominantly in the following form:

```
name: transformation name
lhs:  left hand side of transformation
rhs:  right hand side of transformation
ec:   enabling conditions
dir:  directions
```

Pattern variables in the left hand side of the transformation may be matched against a terminal in the parse tree or against a subtree of the parse tree. The right hand side replaces the subtree against which the left hand side was matched, after substitution of the pattern variables. The enabling conditions are procedures which check the applicability of the transformation. The directions indicate suggested relative places in the program to examine for applying further transformations. In many cases, the directions also suggest specific transformations to try next.

There are fifty transformations in the current version of SPECIALIST which may be conceptually grouped into families as described below.

Language Independent Transformations. These are transformations which derive from properties of programming worlds which are common to many different languages. Although they may be expressed differently in different languages, their content is the same. For example, the mathematical identity X+0 => X may be expressed in the following ways:

```
ALGOL: X+0 => X            (infix language)
LISP:  (PLUS X 0) =>  X    (prefix language)
RPN:   ...X 0 + => ...X    (Postfix language)
FFP:   +.[id,0] => id      (Backus' functional language)
```

Similarly there are transformations which apply to the same control structure in different languages. One example is simplifying a conditional statement when the condition is known or decidable. Another is simplification of conditional or iterative execution of the empty statement.

Language Dependent Transformations. These are transformations which are expressible as properties of the primitives in a particular language -- for example, in LISP we have (CAR (CONS A B)) => A.

Domain Specific Transformations. These are like the language independent transformations, but derive their usefulness from features which are higher level constructions in special knowledge domains. For example, from matrix arithmetic we have the multiplicative identity A*I => A. From number theory we may have gcd(x,x+y) =>

gcd(x,y).

Definitional Transformations. These are transformations useful in SPECIALIST for invoking other transformations. For example, the statement "A is triangular" is a much more convenient shorthand than the complete statement:

A[i,j] => if i ≤ j then 0 else A[i,j]

Dynamic Transformations. Dynamic transformations are used in SPECIALIST as a means for broadcasting relational information within restricted areas of the program. This technique was selected in preference to more computationally complex methods of deduction. This approach was taken as an expedient to getting a working system, but the method is somewhat lacking in generality. To see the utility of this approach, however, consider the statement "for i := 1 until n do S". We can dynamically create two transformations:

i≥1 => true
i≤n => true

and broadcast them throught the corresponding statement S. It is clear, however, that simplifications in S can take place only if expressions in S appear in the right form. At the present time, S would be simplified if it contained a statement of the form "if i≥ 1 then ...", but it would not be simplified if it contained the expression "if i>-100 then ..." or "if i<n+1 then ...". This decision in the design of SPECIALIST represents an arbitrary cutoff in the spectrum of deductive power in arithmetic and boolean simplification. Here is an instance where we would have liked to use the work of other investigators, but could not for reasons of economics.

Transformation Chaining

The SPECIALIST system takes care to limit the space of search for the next transformation to be applied. The "dir" information associated with a transformation is an augmentation to the "pure" production system in that it suggests a search domain and some possible rules to be applied. Assuming that a transformation has been fired, the subtree in which the "lhs" has been matched is called the locus of the transformation. One form of the directions is (here (P1 P2)) or (up (P1 P3)). The first of these indicates that the transformations P1 and P2 are to be tried at the current locus. The second of these says that transformations P1 and P3 should be tried at the level of the tree enclosing the current locus. Another form for the directions is (here (look-at (operators, operands)). This uses a procedure, look-at, which generates a list of recommended transformations from the locus in which the

previous transformation took place. In either case, the recommended transformations are stored in a data structure called the agenda. Suggested transformations are taken from the agenda and attempted (possibly adding more to the agenda) until the agenda is found to be empty.

The directions in the SPECIALIST rules were found to be applications of two meta-rules -- that is, rules which suggest what rules to apply. These can be described formally by assuming that each production Pi has a left hand side Li and a right hand side Ri. A pattern A is said to be a constituent of a pattern B if A can be pattern matched to a subexpression of B. The two meta-rules are then as follows:

> Meta-rule 1: If Pi has just fired and Ri is a constituent of some Lj, then try Pj at some enclosing locus.
>
> Meta-rule 2: If Pi has just fired and Lj is a constituent of Ri, then try Pj at some lower locus.

This concept of guiding the system by suggested chains of transformations has subsequently found acceptance outside of our project. Although the directions for SPECIALIST were compiled by hand, later work by Neighbors has shown that this can be done automatically [Neighbors 1979]. The concept of chaining in SPECIALIST has also found use in the SETL project [Paige 1979].

5. Verification of Pattern-Directed Transformations

In this section we report on our progress in formal verification of pattern-directed transformations.

If each of the transformations Ti has been proven to preserve program equivalence, then by transitivity, any program P' derived from an initial program P by application of a sequence of such transformations, P'= Tn(...T2(T1(P)...)) has been proven equivalent to P. Thus to guarantee that a given transformation catalogue (or program manipulation system) can be used to improve programs while leaving their net effects invariant, it is important to prove, once and for all, that each transformation preserves program equivalence.

The method we describe here is based on Dijkstra's weakest precondition method for program verification [Dijkstra 1976]. We believe that proofs in the Dijkstra formalism are usually clear and easily understood. Since his method subsumes the problem of showing that a program properly terminates, a complete mechanization of his formalism would yield a solution to the halting problem. For our purposes of exposition, we apply these methods manually.

In this section we shall use the following notation for Boolean connectives:

| | "or"
& "and"
-> "implies"
~ "not"
= "logical equivalence"

Oftentimes, particular transformations can be shown to preserve equivalence only if certain enabling conditions are satisfied. Usually the complete enabling condition for a transformation is too complicated to be useful, so we work with enabling conditions that are sufficient -- i.e., their satisfaction guarantees that the transformation is valid although the transformation might be valid in situations where the enabling condition is not met. For example, the following transformation which performs "distribution of multiplication over conditionals" is valid if the program fragments X and Y commute:

```
X*(if Y then Z else W)
   => (if Y then X*Z else X*W)
```

We say that fragments X and Y commute if their order of execution can be interchanged without loss of program equivalence. Using data flow analysis, it will be shown that this is the case if neither X nor Y write into any

variables read by the other, and if they do not write into common live variables.

In Dijkstra's method we associate with each program construct S a predicate transformer, written wp(S,R), which maps an arbitrary predicate R into the predicate wp(S,R). In Dijkstra's own words the meaning of wp(S,R) is "the weakest precondition for the initial state such that activation will certainly result in a properly terminating happening, leaving the system S in a final state satisfying the post-condition R". It is assumed that the predicate R is defined for every tuple of values that the program variables may take on. When we define the semantics of a programming language we associate with each program construct S a predicate transformer wp(S,R) which, by fiat, is the weakest precondition. We assume, as does Dijkstra, that the computation of any predicate is side-effect-free.

Another way of looking at wp(S,R) is to cast its definition in terms of the Hoare formalism. In this context one defines wp(S,R) by requiring that it satisfy the following three constraints:

1. {wp(S ,R)} S {R}
2. if {P} S {R} then P -> wp(S ,R)
3. wp(S,R) guarantees the proper termination of S

Following Dijkstra we say that if the input predicate implies wp(program, output predicate), then the program is semantically valid. We wish to define the semantic validity of a transformation. A block is a sequence of statements with a single entrance and exit. Two blocks S and S' are semantically equivalent if they have the same entrance and exit points and wp(S,R) = wp(S',R) for all predicates R. If a program is semantically valid and it contains a block S which is semantically equivalent to a block S', then the new program formed by replacing S with S' will be also semantically valid. Once the reader is provided with a weakest precondition definition of the semantics of a programming language, and the algebraic rules for manipulating weakest preconditions, he will find this statement easily verifiable. In fact, the new program is semantically valid as long as wp(S,R)->wp(S',R). A transformation T preserves correctness if for all predicates R, wp(S,R) -> wp(T(S),R). A transformation T is equivalence preserving (and hence may be applied in either direction) if for all predicates R, wp(S,R)=wp(T(S),R).

Semantics of Program Constructs

In the following summary, we present some results from Kibler's Ph. D. thesis [Kibler 1978]. We will first define the semantics of some program constructs using weakest preconditions and also give some of the algebraic rules for

manipulating expressions involving the weakest precondition. Given these rules for manipulation we can then show that the result of applying "correctness preserving" transformations is semantically valid (in terms of the original program specifications).

1. Assignment: wp(x:=y,R)=sub(R,x,y),i.e. substitute y for each free occurrence of x in R

2. Concatenation: wp(S;S',R)=wp(S,wp(S',R))

3. Label: wp(L:S,R)=wp(S,R)

4. Empty: wp(empty,R)=R

5. Conditionals:

   a. wp(<u>if</u> B <u>then</u> S <u>else</u> S', R)
      = B&wp(S,R)|~B&wp(S',R)
   b. wp(<u>if</u> B <u>then</u> S, R)=B&wp(S,R)|~B&R

6.Goto: wp(<u>goto</u> L,R)=wp(S,R) where L:S is a uniquely L labelled statement

7. Iteration: wp(<u>while</u> B <u>do</u> S,R)=H(k,R) for some k, where

   H(0,R)=~B&R
   H(k,R)=wp(<u>if</u> B <u>then</u> S, H(k-1,R))|H(0,R)
   =B&wp(S,H(k-1,R))|~B&H(k-1,R)|H(0,R)
   (Note H(k,R) implies H(k+1,R) for all k.)

Algebraic Laws -- These laws can be proven from the above axioms [Dijkstra 1976].

8. wp(S, false)=false

9. if Q->R then wp(S, Q)->wp(S, R)

10. wp(S,Q)&wp(S,R)=wp(S, Q&R)

11. wp(S,Q)|wp(S,R)=(S,Q|R)

<u>Theorem</u>: If P is a semantically verifiable program that contains a block of code S and P' is the program formed by replacing S by T(S) where T is a correctness preserving transformation, then P' is also semantically verifiable.

<u>Proof</u>: To say that P is semantically verifiable means, in our context, that I implies wp(P,O) where I stands for the input predicate and O stands for the output predicate. Decompose P into the sequence of statements P=R;S;Q. The

program P' has the form R;T(S);Q. We must show that I implies wp(P',O). The proof is a simple calculation:

```
I ->wp(R;S;Q,O)                 given
  = wp(R,wp(S,wp(Q,O)))         by multiple application of 2
  ->wp(R, wp(T(S),wp(Q,O)))     by definition of correctness
                                 preservation (R=wp(Q,O))
                                 and algebraic law number 9
  = wp(R;T(S);Q, O)             by multiple applications of 2
  = wp(P',O)
```

If the proof is analyzed one sees that we actually prove that if T is correctness preserving when wp(P,R) implies wp(P',R) for any predicate R. Hence any statement we make about the program variables which will be true after the execution of P will likewise be true after the execution of P', and we may regard P' as an equivalent program.

Clearly an equivalence preserving transformation may be applied in either direction without destroying semantic validity.

Examples of Correctness Preserving Transformations

We are now in the position of being able to verify that some particular transformations are correctness preserving.

Example 1: Simplifying Conditionals. Consider the transformation T on two nested conditional statements:

```
if B then (if A then S1 else S2) else S2

    =>

if B&A then S1 else S2
```

The following direct calculation shows that this transformation is equivalence preserving:

```
wp(if B then (if A then S1 else S2) else S2, R)
                                  by 5a
B&wp(if A then S1 else S2, R) | ~B&wp(S2,R)
                                  by 5a
B&(A&wp(S1,R) | ~A&wp(S2,R)) | ~B&wp(S2,R)
                                  by distribution
B&A&wp(S1,R) | B&~A&wp(S2,R) | ~B&wp(S2,R)
                                  since B&~A | B=~(A&B)
B&A&wp(S1,R) | ~(B&A)&wp(S2,R)
                                  by 5a
wp(if B&A then S1 else S2,R).
```

Other transformations involving simplificaton of conditionals can similarly be verified. If it happens that wp(S,R) is not equal to wp(T(S),R) one can add additional

conditions so that wp(S,R) implies wp(T(S),R). These additional conditions are precisely the enabling conditions mentioned previously. The weakest enabling condition, and least useful, is that wp(S,R) -> wp(T(S),R).

Example 2: Commutativity of Statements. The transformation of S1;S2 into S2;S1 requires that wp(S1;S2,R)= wp(S2;S1,R) for all R. In general there is no way to prove this since, in fact, it is false. We will discuss one instance where S1 and S2 are both assignment statements. Let sub(S, x1,y1,x2,y2,...) denote S where all free occurrences of the xi's have been replaced by the corresponding yi's. Now suppose S1 is x:=x+1 and S2 is x:=x+3. We will now show that these two statements may be commuted. The calculation is:

```
                wp(S1;S2,R)
                            by 2
          wp(S1,wp(S2,R))
                            by 1
    wp(S1,sub(R,x,x+3))
                            by 1
   sub(sub(R,x,x+1),x,x+3)
                            by simple algebra
   sub(R,x,x+4)
```

In a similar manner one checks that wp(S2;S1,R)=sub(R,x,x+4) so that the predicates are equal and the statements may be commuted. If, however, S1 were x:=6*y and S2 were y:=6*x then wp(S1;S2,R)=sub(R,y,36*y,x,6*y) while wp(S2;S1,R)=sub(R,y,6*x,x,36*x). Since these expressions are not equal, the statements may not be permuted.

Example 3: Generation of Enabling Conditions. One transformation listed in the Irvine Catalogue maps the statement "while B do empty" into "empty". Let us compute the weakest precondition for each statement. By 4, wp(empty,R)=R. The computation of the while form is somewhat more complicated in general, but in this instance it is fairly simple. By 7:

H(0,R)=~B&R

Again by 7, with K=1, we have:

H(1,R)=~B&wp(empty,H(0,R)) | ~B&H(0,R) | H(0,R)
=~B&R

by using 4. By a simple inductive argument we find that H(k,R)=~B&R for all k. Hence the weakest precondition for the transformation of the above while form is ~B&R. In order that these two predicates be equal we require that the enabling condition B be false. Now we notice that the original left hand side is actually an infinite loop unless

B is false. Hence, by a straight-forward applicaton of Dijkstra's methods we have discovered an error in the Catalogue, which went unnoticed by many of its readers.

In order to establish the next example it is helpful to take the following definition. If for all predicates R B&wp(C,R) = wp(C,B&R), then we say that B is <u>invariant</u> with respect to C. We assert that if B is invariant with respect to C then so is ~B [Kibler 1978].

<u>Example 4: Movement over Conditionals</u>. Consider the following transformation:

<u>if</u> B <u>then</u> C; <u>if</u> ~B <u>then</u> D

=>

<u>if</u> B <u>then</u> C <u>else</u> D

We now calculate the weakest precondition for each program fragment:

wp(<u>if</u> B <u>then</u> C; <u>if</u> ~B <u>then</u> D,R)
by 2
wp(<u>if</u> B <u>then</u> C, wp(<u>if</u> ~B <u>then</u> D,R))
by 5b
wp(<u>if</u> B <u>then</u> C, ~B&wp(D,R) ¦ B&R)
by 5b, 8, 11 and simplification
B&wp(C,B&R) ¦ B&wp(C,~B&wp(D,R)) ¦ ~B&wp(D,R)

We also have

wp(<u>if</u> B <u>then</u> C <u>else</u> D,R)
by 5a
B&wp(C,R) ¦ ~B&wp(D,R)

These forms are unequal so we need an enabling condition. If we now assume that B is invariant with respect to C, then B&wp(C,B&R) simplifies to B&wp(C,R) and B&wp(C,~B&wp(D,R)) simplifies to wp(C,B&~B&wp(D,R)) which equals wp(C,false) which, by 8, is merely "false". Now the weakest precondition for the entire first fragment reduces to B&wp(C,R) ¦ ~B&wp(D,R), which is precisely the weakest precondition of the second fragment. We have proven that the above transformation is valid under the enabling condition of the invariance of B with respect to C.

<u>Example 5: Transforming the While Statement</u>. Let S1 be the construct "<u>while</u> B <u>do</u> S" and let S2 be the construct "<u>if</u> B <u>then</u> S; <u>while</u> B <u>do</u> S". We wish to prove that these two constructs are equivalent. The proof will be divided into two cases. The predicate B is assumed to be side effect free.

Case 1. Assume B is initially false.

By 7 we have that wp(S1,R)=H(k,R) for some k where we can take

H(0,R)=R&~B

and for k greater than zero,

H(k,R)=B&wp(S1,R) | ~B&H(k-1,R) | H(0,R)

In the case that B is initially false this simplifies to H(k,R)=~B&H(K-1,R) | H(0,R). If we let k be one we see that H(1,R)=H(0,R). By a simple inductive argument we have H(k,R)=H(0,R) for all k. Consequently wp(S1,R)=~B&R. Using this result we can simplify wp(S2,R) to wp(<u>if</u> B <u>then</u> S, ~B&R) which reduces, by 5b, to ~B&R. Hence, in the case that B is initially false, S1 and S2 are equivalent program fragments.

Case 2. We assume that B is initially true.

It suffices to prove that B&wp(S1,R) = B&wp(S2,R). The computation goes:

B&wp(S2,R)
by 2
B&wp(<u>if</u> B <u>then</u> S, wp(<u>while</u> B <u>do</u> S, R))
by 7
B&wp(<u>if</u> B <u>then</u> S, H(k,R) for some k)
by 12
B&wp(<u>if</u> B <u>then</u> S, H(s,R)) for some s
since H(0,R)&B=F
B&([wp(<u>if</u> B <u>then</u> S, H(s,R)) for some s] | H(0,R))
by distribution twice
B&wp(<u>if</u> B then S, H(S,R)) | H(0,R) for some s
by definition of H(s,R)
B&H(s+1,R) for some s
since H(0,R)&B=F
B&H(s,R) for some s
by semantics of the <u>while</u> form
B&wp(S1,R)

This completes the proof for the second case, so the two programs fragments are equivalent and the transformation is valid.

In this section we have seen several examples of verifying the correctness of transformations via the calculation of the weakest precondition. After doing a few computations of this sort, the method seems clear and straightforward. The proofs are short and suggest the necessity of an enabling condition and also the form of the enabling condition itself. The method, therefore, not only

verifies a transformation but is able to suggest the form of the transformation.

6. Program Refinements

In this section we report on our study of the processes and techniques of program refinement.

In the program refinement paradigm, a program is not a single presentation of an algorithm, but rather consists of several statements of the same algorithm at different levels of detail. This technique can be of benefit at various points in the life of the program. It is a natural reflection of a top-down design discipline in that it records the successive stages in the design as successive refinements. It is also useful for the orderly documentation of the program in a way which is easy to comprehend. In the maintenance phase, this decomposition of a program helps the programmer understand how changes fit into the overall design of the system.

We observe that the history of software engineering has shown a progression toward mechanizing greater parts of the design refinement. This began with hexadecimal assemblers, macro assemblers, compilers for algorithmic languages, extensible language systems, and most recently with work on very high level languages. At each step in this progression mechanical techniques were introduced which permitted the programmer to write in terms of higher level abstractions, providing automatic transformation of the abstract program into an executable program. In assemblers and what we conventionally call "high-level language" compilers, the abstractions are fixed into the language and the mappings of features into machine language are mostly rigid. The introduction of optimizing compilers gave rise to an approach which is now central to the idea of very high level languages -- namely, that the compiler can and ought be capable of alternative implementations of the program.

In a well-defined language in which the programmer can write at a very high level, it is highly likely that straightforward implementation of what he writes will lead to inefficiency in the resulting program. As we have observed earlier, the technique of source to source program transformations holds promise for improving mechanically produced programs. In our study of program refinement, we have centered our attention on studying the mechanisms of multi-level program representation without attempting to change program performance qualitatively by mechanical means. We have undertaken to understand how a programmer himself can communicate the nature of a "clever" algorithm, hoping for mechanical assistance only after he has given the system a basic amount of guidance.

We propose that a system supporting multiple levels of program representation and offering mechanical aid in program refinements would offer a number of advantages. Naturally, any part of program development or maintenance

that can be mechanized will be less error prone, can be performed more quickly than by purely manual means (once or many times), and will be much more amenable to checking programming discipline and gathering statistics. We also see multi-level representation of programs as a valuable form of program modularization, with advantages much like those of conventional modularization. In this sense the programmer can separate the high-level design of his program from the details of implementation. Thus any changes to the program which affect only the "details" cannot affect the correctness of the overall program design. We feel that the usefulness of multi-level descriptions for human understanding of programs indicates that this may also be a useful tool for mechanical program verification.

In the next section we give an example of a program refinement, in this case a derivation of the Quicksort algorithm on an array of integers. We then discuss some of the issues which arise in the mechanization of one refinement transformation from the example. We then conclude with some remarks on remaining questions regarding verification and source to source transformations.

6.1 An Example of Program Refinement: Quicksort

This section gives several stages of refinement of the Quicksort subroutine. The Quicksort subroutine is presented as a sequence of refinements, beginning with abstract input and output predicates. The stages of refinement of the Quicksort procedure are as follows:

Q0) The I/O Specification for Sorting an Array of Integers
Q1) Recursive Quicksort with Array Partitioning Asserted
Q2) Useful Quicksort

Furthermore, the refinement from Q1 to Q2 involves the refinement of array partitioning from an action which is merely asserted to an actual algorithm. The refinement of this algorithm, called Quicksort Partition, can be outlined as follows:

P0) The I/O Specification for Quicksort Partition
P1) Partitioning by Partition Element
P2) Partitioning in the Subsequence Micro-World
P3) Partitioning Using Array Operations

Figure 6.1
Q0 -- The I/O Specification for Sorting an Array of Integers

```
INPUT

        A,n

OUTPUT

        A

PRECONDITION

        A: array[1..n] of integer
        n≥1

ALGORITHM

        permute (A) st
            (1≤i≤n-1) implies A[i]≤A[i+1]
```

Figure 6.2
Q1 -- Recursive Quicksort with Array Partitioning Asserted

```
ALGORITHM

        if n>1 then
            permute(A) st
                exists B,C st
                    (B,C) partition of (A) &
                    LEFT(B,C) &
                    not EMPTYSET(B) & not EMPTYSET(C) &
                    (x in B) & (y in C) implies x≤y
            QUICKSORT(B)
            QUICKSORT(C)
```

Figure 6.3
Q2 -- Useful Quicksort

ALGORITHM

```
proc QUICKSORT(A,k,m)
    if (m-k+1) > 1 then
        z := A[1]
        i := k; j := m
     L:
    L1: if i = j then goto T
        if A[j] > z then
            j := j - 1; goto L1
        A[i] := A[j]; i := i + 1
    L2: if i = j then goto T
        if A[i] < z then
            i := i + 1; goto L2
        A[j] := A[i]; j := j - 1
        goto L
     T: A[i] := z
        if j = m then j := j - 1
        else i := i + 1
        QUICKSORT(A,k,i-1)
        QUICKSORT(A,j+1,m)
```

Figure 6.4
P0 -- I/O Specification for Quicksort Partition

INPUT

```
A,n
```

OUTPUT

```
A,B,C
```

PRECONDITION

```
A: array[1..n] of integer
n>1
```

ALGORITHM

```
permute(A) st
    exists B,C st
        (B,C) partition of (A) &
        LEFT(B,C) &
        not EMPTYSET(B) & not EMPTYSET(C) &
        (x in B) & (y in C) implies x<y
```

Figure 6.5
P1 -- Partitioning by Partition Element

PRECONDITION

```
#A = n
A: set of integer
n>1
```

RESULT

```
B,C) partition of (A) &
not EMPTYSET(B) & not EMPTYSET(C) &
(x in B) & (y in C) implies x<y
```

ALGORITHM

```
B,C := {}
z := ARBITRARY ELEMENT(A)
forall y in A--z do
    if y<z then B := B union {y}
    else C := C union {y}
if EMPTYSET(C) then C := C union {z}
else B := B union {z}
```

Figure 6.6
P2 -- Partitioning in the Subsequence Micro-World

PRECONDITION and RESULT

(same as P0)

ALGORITHM

```
"at any time:
    (B,M,C) partition of (A) &
    LEFT(B,M) &
    LEFT(M,C) &
    (x in B) & (y in C) implies x≤y"
let EMPTYSET(B) & EMPTYSET(C)
"during the loop below:
    EMPTYSLOT(RIGHTMOST(M)) or EMPTYSLOT(LEFTMOST(M))"
MOVE TO (z) LEAVING EMPTY (LEFTMOST(M))
loop:
    "EMPTYSLOT(LEFTMOST(M))"
L1: while RIGHTMOST(M) > z do
        MOVE (M,C) BOUNDARY (left)
    MOVE (B,M) BOUNDARY (right)
    MOVE TO (RIBHTMOST(B)) LEAVING EMPTY
            (RIGHTMOST(M))
    "EMPTYSLOT(RIGHTMOST(M))"
L2: while LEFTMOST(M) < z do
        MOVE (B,M) BOUNDARY (right)
    MOVE (M,C) BOUNDARY (left)
    MOVE TO (LEFTMOST(C)) LEAVING EMPTY (LEFTMOST(M))
repeat this loop until M = [the empty slot] at L1 or
        L2
MOVE TO (the empty slot) FROM (z)
if EMPTYSET(C) then MOVE (M,C) BOUNDARY (left)
else MOVE (B,M) BOUNDARY (right)
```

Figure 6.7
P3 -- Partitioning Using Array Operators

ALGORITHM

```
      z := A[1]
      i := 1; j := n
   L:
  L1: if i=j then goto T
      if A[j] > z then
          j := j - 1; goto L1
      A[i] := A[j]; i := i + 1
  L2: if i=j then goto T
      if A[i] < z then
          i := i + 1; goto L2
      A[j] := A[i]; j := j - 1
      goto L
   T: A[i] := z
      if j = n then j := j - 1
      else i := i + 1
```

Representations of Quicksort and Quicksort Partition

Figures 6.1, 6.2, and 6.3 give the representations of the Quicksort algorithm: Q0, Q1, and Q2, respectively. Figures 6.4 through 6.7 give the representations of the Quicksort Partition algorithm: P0, P1, P2, and P3, respectively.

Figure 6.1 gives the Q0 representation of the Quicksort algorithm, showing the INPUT variables, A and n, the OUTPUT variable, A, the PRECONDITION, which consists of declarations relating A and n, and the ALGORITHM. At this stage the algorithm consists of just an asserted statement which says what is to be done, without saying how it is to be accomplished. The statement "permute (A) st (-)" could be taken as a direction to generate all possible permutations until one is found satisfying the desired predicate. Subsequent refinements of Q0 will supply a more efficient procedure than this. Q0 is basically an input/output specification for Quicksort, and is equally valid for any other sort on an array of integers.

Figure 6.2 gives the Q1 representation of the Quicksort algorithm. This gives a decomposition of the Quicksort algorithm into a control structure, which is stated explicitly, and an asserted statement which actually performs the data manipulation of the algorithm. The control structure consists of a manipulation step followed by two recursive calls to the QUICKSORT procedure. The data manipulation is the Quicksort Partition algorithm, which will be refined later. It is possible to decompose the reasoning about Quicksort using the Q1 representation. Assuming that the asserted statement performs as asserted, it is reasonably straightforward to demonstrate the correctness of Q1. This constitutes the first subgoal for reasoning about Quicksort, the second one being the correctness of the refinement of the asserted statement. What we have here is a convenient and intelligible source of guidance for a program verifier.

Figure 6.3 gives Q2, the final representation of the Quicksort algorithm for purposes of this discussion. The differences between Q2 and Q1 are subsumed by the chain of refinements P0 - P3. The Quicksort program at this point could be subjected to any number of improvement transformations to make it run more efficiently. For example, one might wish to replace the recursion with explicit stack manipulation, or one might make changes to balance the size of the partitions B and C at each iteration, and so on. These improvement transformations are beyond the scope of this section which concerns refinements of representation, but the earlier sections on program improvement transformations would clearly come into play here.

Figure 6.4 gives P0, the I/O specification for the Quicksort Partition algorithm. The outputs are A, B, and C. As with Q0, the operation is simply asserted at this level.

Figure 6.5 gives P1, an algorithm for partitioning by partition element. This example represents an abstraction of the actual problem to be solved. In P0 the problem is to partition an array, but in P1 we ignore the array property and consider an algorithm which will work for any set, ordered or unordered. This establishes a "plan" for later refinements and provides the basis for reasoning about later steps.

Figure 6.6 gives P2, a refinement which performs Quicksort Partitioning in terms of a formal abstract world which we call the Subsequence Micro-World. The full input and output assertions have been restored. The following primitive actions and predicates are used:

```
EMPTYSET(-)
RIGHTMOST(-)
LEFTMOST(-)
MOVE TO (-) LEAVING EMPTY (-)
MOVE TO (-) FROM (-)
EMPTYSLOT(-)
MOVE (-,-) BOUNDARY
[ - ]
```

The square brackets are used as a constructor for building a sequence of elements. The operators of this micro-world are described more fully in the next section.

In this program we refer to "the empty slot" without having elaborated the somewhat special notion of "empty" and its relation to the operators above. In fact, we have only implied that it is unique.

The loop termination condition in this program is presented in an interesting way. The body of the loop is written in a way that demonstrates the "typical" or operating cases of the loop, but does not consider the loop termination. The "repeat" phrase at the end of the loop is a meta-program statement which tells one how to modify the text of the program above in order to get a correct version of the program.

It is interesting to note that the data structures at the P2 level need not be considered arrays. It would suffice to make them sequences, since no array operators are used in this algorithm. Doing this would yield the interesting progression:

P1 -- sets
P2 -- sequences
P3 -- arrays

in which we add attributes to the data structures progressively at each step. Figure 4.7 shows P3, the final version of the Quicksort Partition algorithm, which is written in terms of the normal array operations of indexing, reference, and assignment. Due to the "topology" implied by the P2 version of the program, it is not possible to use the normal Pascal style high-level program constructs (while, repeat, etc.). The closest we could come in a pseudo-Pascal would be the following:

```
while (if i=j then goto T; A[j]>z) do
    j := j - 1
```

The refinement from P2 to P3 is quite straightforward, given the introduction of two variables i and j which identify the boundaries between B and M, and M and C, respectively. This is discussed in the next section.

6.2 Mechanical Refinement

In this section we treat some of the issues which we feel bear on the development of automatic program refinement.

The greatest power can be conferred upon the system by giving it a knowledge of the exact programming domain which is most natural for solving the problem at hand. As humans with common education and experience find it easiest to communicate with each other, so it will also be easier to communicate with a system when it has a vocabulary corresponding to a "normal" programmer's education. We believe that to a certain degree this knowledge can be partitioned into micro-worlds, only a few of which would be required for solving any given problem. Note that in the Quicksort example, the various versions of the program referred not only to arrays, but also to sub-arrays, sets, and sequences. However, there was no need for reference to vocabularies for random access files, partial orders, character string scanning, graph theory, and so on. One particular micro-world was very well suited to the P2 representation of the Quicksort Partition algorithm. We may properly call this the Subsequence Micro-World, for it deals with its data structures as ordered sequences (like arrays but without random access) which may be partitioned into disjoint subsequences. A very useful property of this world is that one can speak of the boundaries between subsequences, and can operate on these boundaries as well as the elements of the sequences. We note that this very abstraction is of similiar applicability in describing other elegant algorithms, such as the Boyer Moore string pattern matching algorithm, or the Heapsort algorithm.

The property "empty" derives from another micro-world which we may call the Pseudo Physical Programming World. In this world, data inherits the vocabulary and attributes of physical matter, and variables inherit the vocabulary and attributes of containers. By this distinction we can perform reasoning about conservation of data (which is precisely what is needed for demonstrating that an operation is a permutation). The terminology of this micro-world helps us make explicit a certain ambiguity which arises in the Quicksort example. One can speak of a "partition" when one is speaking of sets and it means that a given element is present in one and only one member of the partition. Formally speaking, then, it is incorrect to speak of partitioning an array which is considered to be a set of values with possible duplicates. When "partitioning" an array it is quite possible that a given element value could occur in two or more partitions. Thus, it is not proper to speak of an array, which can have duplicate elements, as a set. Yet when we confer set terminology on an array it is quite clear what we mean most of the time. The reason for this is that (in this example) we are using set terminology

to refer to the "cells" or "variables" constituting the array, and not to the values they hold. This micro-world, then, helps us deal with a linguistic problem by making this source of imprecision explicit.

A very real problem in identifying and elaborating on these micro-worlds is the tendency toward diversity in the linguistic preferences of programmers. This has become evident with the evolution of higher level languages with extensions designed to be handy for "common" situations.

The use of natural language, or at least semi-natural languages, is another promising technique for elevating the level at which a programmer communicates with the system. Although our case studies such as the Quicksort example have shown the need for this capability, we have not pursued this direction in our research.

It is evident from the examples we have studied that deduction must play a role in the interpretation of high level programs, in addition to being helpful in improving refined programs.

We shall see examples of each of these issues as we consider the P2 representation of the Quicksort Partition algorithm in the remainder of this section.

The Refinement of P2

The first step in refining P2 to a concrete program is to eliminate the "meta program" statement dealing with the loop termination. Since this "meta statement" refers to the places where the loop exit may occur, it is not hard to see how it is to be translated. Without this information in the program, much more sophisticated understanding of the program would be required. With this input however, we can reasonably expect an automatic system to "fix" the program correctly and even to be able to verify that the statements within the while loops always operate on well defined data. The new version of the loop is as follows:

```
 L:
     "EMPTYSLOT(LEFTMOST(M))"
 L1: if M=[the empty slot] then goto T
     if RIGHTMOST(M)>z then
         MOVE (M,C) BOUNDARY (left)
         goto L1
     MOVE (B,M) BOUNDARY (right)
     MOVE TO (RIGHTMOST(B)) LEAVING EMPTY (RIGHTMOST(M))
     "EMPTYSLOT(RIGHTMOST(M))"
 L2: if M=[the empty slot] then goto T
     if RIGHTMOST(M)<z do
         MOVE (B,M) BOUNDARY (right)
         goto L2
     MOVE (M,C) BOUNDARY (left)
```

```
      MOVE TO (LEFTMOST(C)) LEAVING EMPTY (LEFTMOST(M))
      goto L
T:
```

At this point all of the program constructs belong to the micro-world of subsequences except for the expression "M = [the empty slot]". In order to describe how this is to be refined we must first consider the refinement of other constructs in this micro-world. The constructs which need to be implemented are:

```
EMPTYSET(x)
MOVE TO (x) LEAVING EMPTY (y)
LEFTMOST(x)
RIGHTMOST(x)
MOVE (x,y) BOUNDARY (left or right)
x=y        "where x and y are subsequences of a
            sequence w"
```

We must presume that the system has knowledge about refining objects in the subsequence world into a concrete implementation using arrays. (We might also suppose alternative implementations such as using doubly linked lists of elements, or linked blocks of elements.) The first thing which is determined is the representation for the subsequence boundaries. In the P2 refinement, we have the following:

```
LEFTMOST(B)=A[1]
RIGHTMOST(B)=A[i-1]    LEFTMOST(M)=A[i]
RIGHTMOST(M)=A[j]      LEFTMOST[C]=A[j+1]
RIGHTMOST(C)=A[n]
```

The meaning of EMPTYSET(x) is determined from the world of sets, namely:

```
EMPTYSET(x)    =>    #x = 0
```

The world of subsequences must provide the meaning of "#x" (the number of elements of x), which for the constructs of our example are as follows:

```
#B = i-1
#C = n-j
```

The resulting refinement of the <u>let</u> statement is:

```
let i-1=0 & n-j=0
```

which by suitable source to source transformations becomes:

```
i := 1
j := n
```

The statements for moving the boundaries become:

```
MOVE (M,C) BOUNDARY (left)    =>   j := j - 1
MOVE (B,M) BOUNDARY (right)   =>   i := i + 1
```

It now remains to show how "[the empty slot]" can be interpreted. According to natural language usage this presupposes that only one empty slot exists. Some deductive mechanism is necessary in the system in order to associate this reference with "LEFTMOST(M)" at L1 or with "RIGHTMOST(M)" at L2. The micro-world must have the notion of equality for subsequences, which will be that their left boundaries and right boundaries coincide. Considering the situation at L1, and calling "[LEFTMOST (M)]" by the name "x", we can write the following:

```
LEFTMOST(x) = A[i]
RIGHTMOST(x) = A[i]
LEFTMOST(M)=A[i]
RIGHTMOST(M)=A[j]
```

This gives us the following condition for loop exit at L1:

```
if (i=i) & (i=j) then goto T
```

and by suitable source to source transformations we can expect the following, as found in P3:

```
if i=j then goto T
```

Using these techniques it is evident that the refinement transformation from P2 to P3 could reasonably be done mostly if not entirely by automatic means.

6.3 Remaining Issues

In this section we conclude with some remarks on program verification and on source to source transformations.

Verification

We feel that the program refinement paradigm may prove to be of great use in program verification. In the architecture of present program verifiers there is little capability for taking advantage of the structure or design of a program. Rather, the entire program is reduced to a large, unstructured theorem which is then presented to a theorem prover. The modularization offered by a refinement decomposition of a program promises to address this problem by decomposing the verification task into subtasks. We can forsee at least three ways in which refinements can be of use.

Asserted Statement Subgoals. In the Q1 representation, array partitioning was asserted, and was later refined into a concrete algorithm. In this case the verification task can be decomposed into two steps:

1) Prove that the high level program is correct, assuming that the "asserted" statements function correctly as asserted.
2) Prove that the asserted statements, as refined, do in fact satisfy the declared assertion.

Abstract Program Representation. In the P1 example, certain of the properties of the top level program were ignored in order to write an algorithm which achieved some but not all of the desired results. In this case the array property of A was ignored, and the partitioning algorithm was given for an unordered set. In this case the verification task can be decomposed into three steps:

1) Prove the abstracted version of the program, achieving some but not all of the overall desired predicates.
2) Prove that the refinement of the abstract version to a complete version is in fact valid. Hence the properties proved above are still true for the lower level program.
3) Prove the remaining required properties of the lower level program.

Proof by Refinement Transformation. In the P2 example, the refinement to P3 was rather straightforward and the desired data properties (A ending up sorted) were achieved in both versions of the program. In this case the verification task can be decomposed into two steps:

1) Prove the high level program.
2) Verify (possibly trivially) that the refinement from the high level program to the low level program is valid.

Source to Source Transformations

In an earlier section of this report concerning the Irvine Program Transformation Catalogue, we observed that certain transformations could not be formally described in their full generality. We speculate that it may yet be possible to formalize some of these transformations by making them source to source transformations operating on some higher level, more abstract version of the programs in question. At this level all the applicable source programs would be the same, but they would be specialized by refinements into various final forms. If this can be done, then those transformations defined as "examples with discussion" may yet be expressible as pattern directed transformations, or program manipulation procedures, but operating on programs written in an abstract, higher level idiom. This would then suggest that every transformation is most fruitfully attempted at its own preferred level of program refinement.

References

Aho,A.V., and Ullman, J.D. [1973]

The Theory of Parsing, Translation, and Compiling, Vol. II, Prentice-Hall, Englewood Cliffs, N.J.

Backus,J. [1978]

Can Programming be Liberated from the von Neumann Style?, ACM 1977 Turing Award Lecture, CACM 21:8 (August 1978), pp 613ff.

Cheatham,T.E., Townley, J.A., and Holloway, G.H. [1979]

A System for Program Refinement, Proc. 4th Internat. Conf. on Software Engineering, Technical University, Munich, Germany, (September 1979) 53-62.

Christensen,C. [1965]

Examples of Symbol Manipulation in the AMBIT Programming Language, Proc. ACM 20th National Conf., 247-261.

Christensen,C. and Shaw, C.J. [1969]

Proceedings of the Extensible Language Symposium, SIGPLAN Notices 4:8 (August 1969).

Dijkstra,E.W. [1976]

A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N.J.

Galler, B.A. and Perlis, A.J. [1967]

A Proposal for Definitions in Algol, CACM 10:4 (April 1967), 204-219.

Hearn,A. [1971] REDUCE 2: A System and Language for Algebraic Manipulation, Proc. 2nd Symposium on Symbolic and Algebraic Manipulation, N.Y., 128-135.

*Kibler,D.F. [1978]

Power, Efficiency, and Correctness of Transformation Systems, Ph.D. Thesis, Computer Sci. Dept., UC Irvine.

*Kibler, D.F., Neighbors,J.M., and Standish, T.A. [1977]

Program Manipulation via an Efficient Production System, Proc. Symp. on A.I. and Programming

Languages, ACM SIGART and SIGPLAN, SIGPLAN Notices, (August 1977), 163-173.

Knuth,D.E. [1967]

The Remaining Trouble Spots in Algol 60, CACM 10:10 (October 1967), 611-617.

Knuth,D.E. [1974]

Structured Programming with Go To Statements, C. Surveys 6:4 (December 1974), 261-301.

Leavenworth, B.M. [1966]

Syntax Macros and Extended Translation, CACM 9:11 (November 1966), 790-793.

Loveman, D.B. [1977]

Program Improvement by Source to Source Transformation, JACM 24:1 (January 1977), 121-301.

Markov,A. [1954]

Theory of Algorithms, Tr. Mat. Inst. Steklov, XLII.

McIlroy, M.D. [1960]

Macro Instruction Extensions of Compiler Languages, CACM (April 1960), 214-220.

McIlroy, M.D. [1969]

Alternatives to Extensible Languages, Proc. of the Extensible Language Symposium, SIGPLAN Notices 4:8 (August 1969), 50-52.

Naur,P. et al [1963]

Revised Report on the Algorithmic Language Algol 60, CACM 6:1 (January 1963), 1-17.

Neighbors, J.M. [1979]

The Construction of Software from Software Components, Ph.D. thesis (in preparation), Computer Sci. Dept., UC Irvine.

Paige, R. [1979]

Expression Continuity and the Formal Differentiation of Algorithms, Computer Sci. Dept., Courant Institute, NYU, (September 1979).

Sammet, J.E. [1969]

Programming Languages: History and Fundamentals, Prentice-Hall, Englewood Cliffs, N.J.

Squires, S. [1979]

Experience with an Advanced Programming Environment in an R&D Setting, Proc. Ada Environment Workshop, San Diego, California (November 27-29, 1979).

*Standish, T.A. [1977b]

Program Transformations, Proc. 1977 Conf. on Info. Sciences and Systems, Johns Hopkins Univ., (March 1977), 326-332.

*Standish,T.A., Harriman,D.C., Kibler,D.F., and Neighbors, J.M. [1976]

The Irvine Program Transformation Catalogue, Computer Sci. Dept., UC Irvine, (January 1976).

*Standish, T.A., Kibler, D.F., and Neighbors, J.M. [1976b]

Improving and Refining Programs by Program Manipulation, Proc. ACM Annual Conf., 509-516.

*Standish,T.A., Kibler, D.F., Neighbors, J.M., and Winkler, T.C. [1977]

Verifying Source to Source Transformations, Computer Sci. Dept., UC Irvine.

* Note: References with asterisks were supported by the NSF Grant reported on in this Final Report.