MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 962 May 1987

Using Program Transformations to Improve Program Translation

by

Thomas R. Kennedy, III

Abstract

Direct, construct by construct, translation from one high level language to another often produces convoluted, unnatural, and unreadable results, particularly when the source and target languages support different models of programming. A more readable and natural translation can be obtained by augmenting the translator with a program transformation system.

Copyright © Massachusetts Institute of Technology, 1987

This report describes research done at ROLM Corporation and at the Artificial Intelligence Laboratory of the Masachusetts Institute of Technology. Support for the Artificial Intelligence laboratory's artificial intelligence research has been provided in part by the IBM Corporation, in part by the Sperry Corporation, in part by the National Science Foundation grant IRI-1811644, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the policies, either expressed or implied of, of the ROLM Corporation, of the IBM Corporation, of the Sperry Corporation, of the National Science Foundation, or of the Department of Defense.

Chapter One

High Quality Translation

1.1 Introduction

In many cases translating programs to more structured, better supported, and/or more strongly typed languages can provide significant savings in maintenance and support.

However, in order to maximize the benefits of translating an existing program into a new language, not only must the translation be correct, it must also be of high quality. A better way to characterize this objective is to say that the output of the translator should be as close as possible to what a human programmer would have written if he/she had originally written the program in the target language. That is, the new code should make use of any data and control constructs available in the target language which were not available in the source language. Similarly, those constructs of the source language which do not have direct analogs in the target language should be expressed in a way natural to the target language.

Unfortunately, direct automatic translation from one high-level language to another is difficult. The easiest solution is to mimic the data and control structures of the source language, often times circumventing the type-checking mechanisms and/or neglecting the control abstractions provided in the target language. The result is code which exhibits all the drawbacks of the source language and few of the advantages of the target.

ROLM Corporation has implemented a translator for translating from RPL (a language similar to C) to PL.8 (a derivative of PL/I). The translation is performed

by first parsing the RPL source and building appropriate symbol tables. Then, each construct in the source is replaced by an equivalent construct (or set of constructs) in the target language in a manner akin to macro-expansion. The final pass outputs PL.8 code.

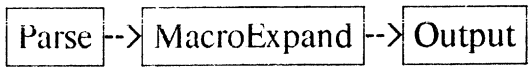

Figure 1-1: The ROLM Translator

Although more or less effective, this approach unfortunately produces PL.8 code which is unnatural, and occasionally incorrect.

The goal of the research reported here is to show that the addition of a program transformation module (PTM) to the translator can provide a markedly improved output.

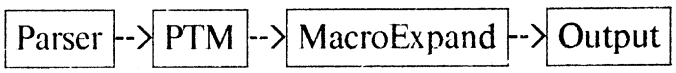

Figure 1-2: Adding the PTM

Program transformations can be used to remove source constructs which cannot be expressed naturally in the target language, and to combine primitive constructs into more elegant expressions available in the target language. Output can be obtained which is both more natural and more correct than that produced via macro-expansion alone.

This approach can yield significant improvement in the quality of a translation compared to macro-expansion alone. However, the technique still suffers from the lack of an abstract understanding of what a program does.

1.2 Translation by Macro-Expansion

RPL is a language similar to C, developed and used internally at ROLM. PL.8 is a derivative of PL/I developed for internal use at IBM.

The overall structure of the two languages is very similar. Procedures comprise the main unit of control abstraction in both languages. Rules for invocation and definition of procedures are the same in both languages. The rules of scope for the two languages are the same (see Section 2-1).

The task of translation is simplified greatly by the structural similarities between the two languages. As a result, translation can concentrate on the local constructs of the languages, mapping each individual construct into an equivalent one in the target language.

However, there are certain local constructs in RPL which do not have direct analogs in PL.8. Similarly, PL.8 offers some abstract constructs which do not exist in RPL. Consider as an example the procedure in Figure 1-3. Designed to serve as a compact example, the procedure executes the body of the loop eleven times, initializing the array elements V[0] through V[10] with the values 1 through 11. At the bottom of the loop a test is performed and the variable J is set to I when the input parameter C has been decremented to 4.

The code contains increment and decrement expressions. These are very common constructs in RPL. However, PL.8 does not support increment or decrement operators (nor does it allow assignments within expressions, another common RPL feature).

PL.8 does, however, offer several loop constructs in addition to the simple loop and test construct which RPL offers.

```
PROC TEST (INT C);

  INT I, J, V[10];

  I = 0;
  LOOP;
    WHILE I <= 10;
    V[I++] = I;
    IF 4 == --C THEN
      J = I;
    ENDIF;
  REPEAT;

ENDPROC;
```

Figure 1-3: Example input to translator.

Still, a translation can be obtained by using simple macro-expansion techniques. The increment and decrement expressions can be replaced with calls to appropriately pre-defined procedures (i.e. post-increment is replaced with a procedure which takes an argument by reference, increments that argument, and returns the original value of the argument). The RPL loop construct can be imitated by using the PL.8 loop construct which matches it.

Using such techniques, Rolm's translator produced the output in Figure 1-4.

The output is a legal PL.8 procedure. Given the appropriate definitions of @AINC16 (**a**fter, **in**crement, **16** bit object) and @BDEC16 (**b**efore, **dec**rement, **16** bit object) (see Section 2-2) the program will compile and plausibly performs the same function as the original RPL code.

While this translation appears to satisfy the primary concern of producing

```
TEST: PROC ( C )  EXPOSED;
DCL C INTEGER HALF  VALUE ;
DCL I INTEGER HALF ,
    J INTEGER HALF ,
    V(10) INTEGER HALF ;

  I = 0;
  DO WHILE (I <= 10);
    V(@AINC16(I)) = I;
    IF 4 = @BDEC16(C) THEN
         J = I;
  END DO;

RETURN;
END  TEST;
```

Figure 1-4: A literal translation of figure 1-3.

equivalent PL.8 code, it is not a high quality translation. This is not the code that a human programmer would have written had he/she originally written this procedure in PL.8.

For example, the procedures @AINC16 and @BDEC16 are not natural PL.8 constructs. A PL.8 programmer would have written separate assignment statements rather than introduce procedure calls at those points.

A PL.8 programmer also would have represented the loop differently, using:

```
DO I = 0 TO 10 BY 1; ... END DO;
```

instead of:

```
I = 0; DO WHILE (I <= 10);...@AINC(I)...END DO;
```

The former choice makes the function of the loop explicit, while the later obscures its purpose.

Even though this ouput is far from optimal, it might be acceptable as long as the

translated version always performed the same function as the original program. However, due to a subtle difference between RPL and PL.8 this is not always the case. The macro-expansion considers assignment in RPL to be equivalent to assignment in PL.8. It is not.

The order of evaluation in an assignment differs between RPL and PL.8. In RPL the value (right-hand side) is evaluated before the target. In PL.8 the target (left-hand side) is evaluated before the value. Usually, this discrepancy does not cause a problem since the target is often a fixed memory location. However, in certain statements, the order of evaluation is crucial; consider

```
V[I++] = I;   vs.   V(@AINC16(I)) = I;
```

If I is 0 then evaluating the RPL statement results in setting V[0] to 0 and setting I to 1. Evaluating the "equivalent" PL.8 statement results in setting V(0) to 1 and I to 1. So in the procedure above, the RPL version initializes the elements of the array to 0 through 10, while the PL.8 version initializes the elements of the array to 1 through 11.

The only way to translate assignment statements via macro-expansion and ensure proper evaluation would be to introduce temporary variables to force proper evaluation, i.e.:

```
@Temp01 = I;
V[@AINC(I)] = @Temp01;
```

However, expanding each assignment into two assignments is not a very pretty solution. Rather than allow a proliferation of temporary variables, the designers of the ROLM translator chose to accept an occasional incorrect result. This sort of trade-off is not uncommon in systems relying solely on macro-expansion.

1.3 Adding Program Transformations

The quality of the translation can be improved by adding a program transformation module. Transformations can be used to identify and remove constructs of the source language which are not naturally expressed in the target language. Similarly, they can be used to identify patterns in the source language which can be mapped into more elegant constructs in the target language.

The transformation module can be placed either before the macro-expansion (transforming the RPL code to remove those constructs that do not translate well), or it can be placed after the macro-expansion (transforming the PL.8 code to produce better output).

The later choice might seem the most obvious. However, often the macro-expansion obscures significant details that were readily apparent in the original code. For instance, turning side-effect operators into procedure calls changes an explicit operator into a construct whose function is only implied by the name of the called procedure. These problems are avoided by placing the program transformation module (PTM) before the macro-expansion.

In order to demonstrate the effacacy of this approach, a program transformation module was implemented and added to ROLM's translator as shown in Figure 1-2. In addition a set of transformations was constructed which remove side-effect operators from within expressions, and to recognize specialized loop forms. The PTM augmented translator is able to the original RPL procedure in Figure 1-3 into the PL.8 code in Figure 1-5.

A number of separate transformations are required to achieve this result (See Section 3-3). First, side-effect operations (such as increment, decrement or assignment) which appear within an expression, are moved into separate assignment

```
TEST: PROC ( C )  EXPOSED;
DCL C INTEGER HALF  VALUE ;
DCL I INTEGER HALF ,
    J INTEGER HALF ,
    V(10) INTEGER HALF ;

  DO I = 0 TO 10 BY 1;
    V(I) = I;
    C = C - 1;
    IF 4 = C THEN
        J = I + 1;
  END DO;

RETURN;
END  TEST;
```

Figure 1-5: A better translation of the procedure in figure 1-3.

statements. This is relatively straightforward in many cases. However, when the side-effect is deeply embedded in an expression, the problem becomes difficult to automate without introducing temporary variables.

The degree of difficulty depends on the meaning of the side-effect expression itself, as well as the meaning of the expressions which surround it. It is essential to know what is and is not affected by the movement of certain operators.

Consider the statement:

```
J = V[I]++ - V[C];
```

The obvious solution:

```
J = V[I] - V[C];
V[I] = V[I] + 1;
```

is not necessarily correct. What if I and C are equal? If so, in the first statement, J is assigned -1 and in the second, J is assigned 0. The transformations take this into account and arrive at the solution:

```
V[I] = V[I] + 1;
J = V[I] - 1 - V[C];
```

In this case, there was a solution. However, when procedure calls or pointer dereferences are involved, it becomes less likely that a reaonable transformation can be achieved. The problem is that there is not enough information about the procedure or object to know how it is affected. In such cases, there is no transformation which can legally be applied, so the side-effect operator is left to be translated via macro-expansion.

Loop transformations are performed when the RPL loop matches a known pattern. In this case the pattern below:

```
LOOP;
  WHILE  <var> relational_op  <exp1> ;
  <loop body>;
  <var> = <var> + <exp2>;
REPEAT;

  where: <exp1> is not modified by <loop body>
         <exp2> is not modified by <loop body>
```

A number of transformations are performed on the original loop in order to match this pattern. First, V[I++] = I is split into two statements, and then I = I + 1 is moved to the end of the loop (resulting in the appearance of J = I + 1 in the if statement).

When the RPL loop matches the pattern above, it is transformed into a complex PL.8 loop as shown in Figure 1-5. The initial and final values of the loop are derived from <var>, <exp1>, and the relational operation used in the test. In this case the initial value is I and the final value is 10, since the test is <=. Had it been <, the initial value would have been I and the final value would have been 9. The initialization of I to 0 before the loop is combined with the PL.8 loop by a separate transformation.

1.4 Results

The translator and PTM describe above are not just laboratory prototypes. They have been tested on existing code drawn from a real time system written in RPL.

Currently, using a set of 150 transformations, the PTM is able to remove 98% of the side-effects within expressions. In most of the remaining side-effect expressions, no legal transformations could be made without introducing temporary variables to insure correct evaluation (e.g. the side-effect appears in an expression between two procedure calls). Close to 80% of the loops in the test code were recognized as specific cases of the specialized PL.8 loop constructs and transformed accordingly.

The addition of the PTM had a minimal impact on the run-time performance of the translator. For 80% of the files tested, the increase in run-time was less than 10%. (Much of the run-time was spent in the I/O bound operations of parsing and output.) In the remaining 20% the increase ranged from 10 to 50%, except in one file for which the increase was 100%.

The PTM approach is by no means a panacea. One can contrive examples on which the system will perform no useful work. However, such examples do not often appear in actual RPL code, typically because they involve expressions which are too complex to readily decipher and are therefore avoided by RPL programmers.

The PTM has been incorporated as a standard part of the translator, and plans are being discussed for adding transformations to implement additional features in the translation.

1.5 Alternative Approaches

The use of program transformations is not new. However, in the past, they have been used primarily for optimization or program synthesis, not for code translation.

Using a program transformation module to make local alterations of a parse tree is not the ultimate solution to the problem of code translation. The approach is limited to making semi-local incremental improvements. Desirable transformations are often limited by a lack of global information. The approach is very similar to a programmer sitting down and looking at a small piece of code and asking how he can perform the same steps in the target language. A real "expert" programmer would sit down and ask what the code does, and how he can perform the same function in the target language. The ultimate solution lies in looking at the bigger picture, determining the functionality of a system and implementing the functionality using the constructs available in the new language [13].

Chapter Two

Translating from RPL to PL.8

The success of the PTM augmented RPL to PL.8 translator relies heavily on the fact that the two languages match closely in several important areas. There are a substantial number of constructs in RPL which can be mapped directly into constructs available in PL.8. This allows the PTM to concentrate on those few constructs which are not naturally expressed in the target language, without having to worry about achieving a complete solution.

2.1 Similarities

2.1.1 Procedures and Scoping

The overall structure of the two languages is very much the same. The rules of scope for each language are similar. Most significantly, both languages allow for static scoping of procedure definitions. That is, they permit the definition of nested procedures such that a nested procedure has access to the local variables of it's parent procedure, and the nested procedure is only known within the scope of the parent procedure. This was a major factor in selecting PL.8 as a target language, rather than C (which does not permit nested procedures).

The basic unit of scope in both languages is the procedure. Neither language introduces a new scope for a block of code. That is, "begin ... end" does not define a new scope in which identifiers may be defined, nor is a new scope introduced for the body of a loop (this implies that the iteration variable within a loop can be initialized prior to the loop and is accessible after the loop terminates).

Both languages offer procedures as the main unit of control abstraction. In both languages a procedure may take zero or more arguments and can return zero or one scalar values. In RPL all parameters to a procedure are passed by value. In PL.8 parameters may be passed either by value or reference.

2.1.2 Scalar Data Types

There is also a significant match between the data types manipulated by the two languages. Both languages support 1 bit, 8 bit, 16 bit and 32 bit data objects (bit, byte, int, and int32 (or pointer) in RPL, bit(1), char, integer half, and integer (or offset) in PL.8). If the basic size of these data objects differed, the translator would need to consider how an object was used when deciding what to map different objects into. Since the sizes match, objects can be automatically mapped into objects of the same size without considering how the size might affect the information it represents.

2.1.3 Arrays and Structures

Both languages support aggregate data types in the form of arrays (homogeneous, accessed by integer indexes) and structures (heterogeneous, accessed by named components). The syntax of structures differs between the two languages, but the semantics is the same.

The rules for disambiguating partially qualified references differ in the two languages. This is dealt with by determining the correct reference using the RPL rules and then expanding it so that it is fully qualified and unambiguous in the PL.8 output.

Also, the specification of initial values for static structures differs in the two languages. In RPL initial values can be given separately for each element of a

structure, or they may be listed consecutively at the end of the data definition. A structure type may also be defined with default initial values. In PL.8 all initial values are declared with the elements of the structure (not grouped at the end), and no default initialization of types is available. Translation from RPL to PL.8 is simply a matter of determining the initial values of a static object (from the type defaults and the trailing initial values) and putting those values on the appropriate structure elements. The actual initialization of static data is the same, and initialization of non-static data is not allowed in either language.

2.1.4 Pointers

RPL allows pointers to memory. The pointers are simply 32 bit integers which can be manipulated as any other integer. These pointers can be de-referenced to obtain the object pointed to (its type is determined by the type declaration of the pointer, or explicitly through a pointer operator). RPL supplies an address-of operator for obtaining the address of any variable, procedure, or label in the current scope.

PL.8 supports references via offsets into data areas. Usually the data area is a specific portion of memory. However, an offset can be based in "$Memory" making it equivalent to a pointer. Offsets can be manipulated as integers. Offsets do not have base types associated with them, de-referencing requires a pointer operator which explicitly states the type of the object being referenced. The address of an object may be obtain through the PL.8 built in function MLOC.

2.2 Differences

2.2.1 Primitive Constructs

A major point of failure in source-to-source translators involves primitive constructs which are permitted in the source language but which do not exist in the target. There are two primitive features of RPL which are not available in PL.8.

The first of these RPL features allows the programmer to declare register variables. These are scalar quantities which are maintained in a CPU register. The programmer declares a register variable when he/she wants to make sure that access to a variable is fast (such as the loop counter in an inner loop). Use of register variables usually stems from a lack of faith in the register allocation scheme of the RPL code generator. Semantically, they are equivalent to other scalar variables, so they can be represented in PL.8 as regular variables.

The second "feature" of RPL has no PL.8 equivalent. Static data declared and initialized inside an RPL procedure is placed in exactly the position it appears, and no jump instruction is generated around that data. Thus, static data declared in the middle of a list of statements will be interpreted as machine instructions and executed. This, along with the register variable facility allows the programmer to insert machine instructions inline whenever he/she is worried about the efficiency of the code generator. There is no way to duplicate this feature in PL.8, and it is not clear that one would want to. Where it does appear, the translator can only flag it as an untranslatable. That portion of the code must then be re-written and/or translated by hand. Fortunately, most RPL programmers have been wise enough to avoid such constructs.

2.2.2 Inter-module interfaces

There are other differences between the two languages which, while they complicate the process, are not insurmountable.

The type checking constraints of RPL are much looser than those of PL.8, in particular, when it comes to procedural interfaces. In RPL, when a procedure call is made, neither the number of arguments, nor their types are checked against the procedure declaration for correctness. Presumably any errors are detected as bugs at run-time. In PL.8 the arguments must match the expected types of the parameters. As a result, in RPL, external procedures merely are declared by name and return type. Neither the number of parameters to the procedure, nor their types is included. In PL.8, an external procedure declaration requires a specification of the parameter list in addition to the procedure name and return type.

In order to obtain the necessary information for external declarations an entire system of files to be translated is run through a portion of the translator to build up a global data-base of procedures, identifiers, and types for the given system. A series of analysis programs are then run on this global data to identify any problems that might arise. Once identified these problems usually require human intervention.

2.2.3 Increment, Decrement, and Assignment

The two languages also differ in a number of local constructs.

As indicated in the previous chapter, RPL provides increment and decrement operators, as well as permitting assignments within expressions. PL.8 does not offer any direct equivalent of these constructs.

These operations can be duplicated by appropriately defined procedures. In order to duplicate the five constructs (pre-increment, pre-decrement, post-increment, post-decrement, and assignment) at least fifteen procedures must be defined -- one for each size data. For example, @AINC16 is defined as a procedure which increments a 16 bit object after returning its value. (The @ sign is not valid in RPL,

so it is used in the PL.8 procedure names to avoid conflicts with RPL identifiers.) @BDEC32 is a procedure which decrements a 32 bit object before returning its new value. @ASSIGN8 assigns the 8 bit value of its second argument to its 8 bit first argument, and returns that value. Actually, the RPL increment and decrement functions are a little bit more sophisticated than that. Most objects are incremented or decremented by 1. However, if the object is a pointer, then it is incremented by the size of the object to which it points. This requires two additional procedures, @AADD32 and @BADD32. Thus if PS is a pointer to a structure XYZ then:

```
++PS
```

becomes:

```
@BADD32 (PS, SIZEOF (XYZ))
```

This solution to translating the side-effect operators, though not pretty, might be acceptable if not for one additional discrepancy between RPL and PL.8. The order of evaluation in assignment differs between the two languages. As shown in the previous chapter a different order of evaluation in the presence of side-effects can be particularly troublesome. One way to correctly translate RPL assignment into PL.8 assignment, would be to split each assignment into two assignments using a temporary variable, thus guaranteeing correct evaluation. However, since in most cases substituting PL.8 assignment for RPL assignment is correct, the prettier solution was preferred even at the risk of an occasional incorrect result.

2.2.4 Loops

Most effort in translation is concentrated on those areas of the source language which cannot be easily translated into some form in the target. An area that is general neglected, but important for high quality translation, is the constructs of the source language which can easily be mapped into a general form in the target, but might be better expressed in a different form.

For example the RPL loop construct:

```
LOOP;
  ...
  WHILE <expression>;
  ...
REPEAT;
```

in its most general form, maps directly into PL.8 as:

```
DO WHILE (TRUE);
  ...
  IF NOT (<expression>) THEN LEAVE;
  ...
END DO;
```

Or, if the test in the RPL loop appears either at the beginning or end of the loop, it can be more elegantly expressed as a while or until clause:

```
DO WHILE (<expression>); ... END DO;

DO UNTIL (NOT (<expression>)); ... END DO;
```

However, this mapping neglects two other PL.8 loop types. The "for" loop:

```
DO <var> = <exp> TO <exp> BY <exp>;
```

in which a variable is incremented by a constant amount each time through the loop until a bound is reached, and the repeat loop:

```
DO <var> = <exp> REPEAT <exp>;
```

in which a variable takes on a new value each time through the loop (usually based on its current value). A high quality translation would make use of these additional loop types where appropriate, rather than relying solely on the general form.

Chapter Three

The Program Transformation Module

3.1 General Scheme

In order to improve the quality of translation a program transformation module was added to the translator. All transformation systems have a set of transformations which they can apply. They differ however in the manner in which particular transformations are selected and applied.

There are a number of ways these transformations can be specified, stored, selected, and applied. The approach used by the PTM was derived by observing how a human might attempt to remove side-effect expressions or replace primitive loop constructs. A human would look through the code, until an "interesting" construct was encountered. Upon encountering an interesting construct, one would then try to replace it with a more appropriate form. For instance, if a loop is encountered, one would attempt to replace it with a PL.8 loop type. If a side-effect is found in an expression, one would attempt to replace the statement it appears in with a series of statements not containing side-effect expressions. Essentially a specific goal is established only after a certain pattern is encountered. That goal is then satisfied in a backward chaining manner.

Within the PTM there is a small set of forward chaining transformations. The PTM walks through the parse tree attempting to apply these transformations to each node in the tree. When a forward transformation matches a node, its consequent is executed. Typically, this means a goal is asserted. When a goal is asserted, an attempt is made to satisfy it by a large set of transformations in a goal directed (backward chaining) manner. By establishing an appropriate specific goal for a situation the work can be directed toward more promising avenues.

Goal directed application does not appear to be common among transformation systems. Forward chaining is more common. One approach is to divide the transformations into small sets. Each set is then applied to the parse-tree in exhaustive post-order manner. That is, for each node, the transformations are applied first to the children of the node, and then to the node (post-order). Each time a transformation is triggered the set is reapplied to the result until no further transformations can be made (exhaustive). This scheme has been used to transformation pure applicative Lisp into equivalent Fortran [3, 4]. By carefully ordering the sets of transformations, the transformation proceeds in logical steps toward the goal.

A primarily backward chaining mechanism was selected over forward application for several reasons. First, it was felt that the conditions of applicability for certain transformations might be arbitrarily complex. In pure forward chaining environment, these applicability clauses would either need to have been calculated in advance (by an earlier set of transformations) or determined on the fly by special purpose code. Given that there may be a large number of conditions, only a few of which might be interesting for any particular construct, the former choice wastes a significant amount of computation. Given that the conditions could be fairly complex, the latter choice is also undesirable. It is better to express the complex ideas using transformations, rather than burying the complexity in a piece of code. At a minimum, there should be some backward chaining facility that can be evoked for the satisfaction of applicability clauses.

Asserting a goal for a particular construct and then satisfying (or failing to satisfy) that goal concentrates the entire effort on a local area. Concentrating on one area at a time, instead of one set of constructs, offers two advantages.

First, it is helpful in debugging the transformations. If a particular construct is

transformed in an unexpected manner, one can examine the entire series of transformations attempted on the construct, without worrying about extraneous applications to other constructs.

Second, in this particular case, the structure of the parse tree was already defined. The parse tree definition was not well suited for program transformations. In order to simplify the handling of nodes, interesting nodes were copied into a more uniform representation in a separate space. By concentrating efforts on one area at a time all transformations can be performed in the preferred representation and the result can be translated back, without maintaining a separate representation of the entire tree.

3.2 Transformation Language

The ROLM translator was implemented in PL.8. The parse tree manipulated by the translator is a PL.8 data structure. As a result, the program transformation module was also implemented in PL.8. The structure of the parse tree had a major influence on the representation of transformations, and their specification.

Each node in the parse tree contains a wealth of information. First, a node contains a type indicator. (Typical node types are if, loop, assign, plus, declare, identifier, constant, dereference, etc.) Each node also has up to four child nodes, designated left, middle, right, and list. These can be null or point to additional nodes. The meaning of each child is dependent upon the parent's node type and its position. For instance the operands of a binary operator, such as plus or times, typically appeared as the left and right children of the node. In an "if" node the conditional expression appears as the left child, the "then" clause is the middle child, the "else" clause is the right child, and the statement following the "if" is its list child.

In addition, a node may contain a pointer to a symbol table entry (for identifier nodes), a value (for constant and string nodes), pointers to surrounding comments (the comments are preserved so they can be included in the translated code), information about where the node appeared in the source file, and numerous flags.

The transformation language was defined in order to describe specific patterns in this parse tree. A transformation consists of a predicate and a consequent. The primary component of the predicate is a pattern which essentially a piece of a parse tree. Each node in the predicate pattern specifies the node type, and four child nodes.

The node type can be specified as one of the specific types which occur in the tree, or it may be specified as a class of types. A hierarchical classification of node types is defined to permit writing a single general transformation to cover a class of patterns, where a set of rules, one for each node type in the class might otherwise be required. This classification system is also used to select and order the transformations for application.

The children of a node pattern can be specified by additional node patterns, by "any" (anything will match), or by "nil" (the child must be null).

Using this scheme, one can specify the general structure of a pattern, but not specific details such as the type of a variable, or the value of a constant. In order to allow the specification of such details, the pattern language was extended to permit the addition of an arbitrary list of additional constraints on each node in a pattern. Each constraint takes the form of a name-value pair. The name is an arbitrary string denoting some attribute of the node, and the value may be a constant (integer or string) or a variable which might have additional constraints on it. These attribute names can be defined with associated PL.8 code for retrieving the value of that

attribute from a given node. For instance, the attribute "const–value" has code associated with it for retrieving the value of a given constant node. Or, if an attribute does not have code associated with it, the constraint can be satisfied by another transformation which asserts an attribute-value pair.

The consequent of a transformation can do one of several things. If it is a forward-chaining transformation, the consequent might assert a goal. Otherwise, a transformation may either specify a replacement pattern for the pattern in the predicate, or it may assert an attribute-value pair on the top node in the predicate.

Replacement of a pattern with another is required to transform the parse tree. Assertion of arbitrary name-value pairs by transformations, permits the use of transformations to define the meaning of new attributes.

Internally, the transformations are represented as a pair of tree structures. A number of PL.8 procedures were defined to permit easy specification and construction of the transformations. The example below is the sequence of PL.8 instructions used to specify a simple transformation:

```
#IF;
#NODE (A is nd_pre_decr, nil, nil, B, C);
   #NODE (B is nd_variable_ref);
      #HAS (Attr_Var_Type, "integer");
#THEN;
#REPLACE (A, F);
   #NODE (F is nd_Assign, B, nil, G, C);
       #NODE (G is nd_Arith_Add, B, nil, H);
            #NODE (H is nd_Constant);
                #HAS (Attr_Const_Value, 1);
#END;
```

Rather than burden the reader with the particular idosyncracies of the definition of parse tree nodes, such transformations will, whenever possible, be expressed in a more familiar fashion:

```
?B++; ?C...   ==>  ?B = ?B + 1; ?C...
  Where: ?B is a variable of type integer;
```

While such paraphrasing may not be precise it is intended to convey the meaning of the transformations involved.

3.3 Transformation Selection and Application

Given a specific goal there are often several transformations which might satisfy that goal. The performance of a transformation system depends on which transformation is chosen in such situations. Haphazard selection of rules can lead to undesirable results or long run-times.

The strategy used for selection in the PTM was that of selecting the transformation most specific to the current goal first, and then trying more general transformations later.

Generality is measured against the class hierarchy. For example if the current goal is to transform an addition into a subtraction, the transformations which deal specifically with addition will be chosen before the transformations which deals with binary operators.

This strategy was adopted based on the reasoning that if both a specific transformation and a more general case of that transformation exist, one should select the specific case when it applies (otherwise, why have the more specific transformation at all?).

In those cases where there are two or more transformations which are equally specific, selection is performed on the basis of the order in which they were originally defined. This permits/requires some hand ordering of rules to produce efficient or desirable results.

The transformation selection strategy is not optimal and occasionally requires back-

tracking. As a simple implementation, partial chronological backtracking occurs when a choice fails to satisfy the goal.

While dependency directed backtracking would be a more efficient implementation, the performance of the chronological backtracking system turned out to be adequate. By using a small number of forward chaining transformations to assert specific goals on certain nodes, the scope of operation is limited significantly. With that reduced scope, the inefficiencies of the rule selection and application scheme do not have as great an impact.

3.4 Selected Transformations

A selected set of transformations appears in Figure 3-1. To see how they are applied, consider the statement:

```
A = I++ + I;
```

When the PTM encounters the post-decrement expression, the forward transformation A1 is triggered. This establishes the goal of replacing the single assignment with a list of statements. Triggering the backward-chaining mechanism.

The only transformation (in the sample set) which transforms an assignment into a list of statements is B1. This transformation expects the right-hand side of the assignment to be a generalized-side-effect expression. A generalized-side-effect (gse) expression is a triple consisting of a list of pre-statements, an expression, and a list of post-statements. For instance, the generalized-side-effect expression for

```
--J + I++
```

would be

```
gse((J = J - 1), (J + I), (I = I + 1;))
```

Transformation B4 will turn a binary arithmetic expression into a gse. However, it requires that the operands of the binary operator be gse expressions themselves.

Figure 3-1: Selected Transformations

A: A Forward transformation, triggered by the presence of an increment or decrement operator within a statement:

```
A1: IF: ?A is a side-effect-class-op
        within-statement ?B
    THEN:
      ASSERT_GOAL: ?B ==> list of statements;
```

B: Transformations dealing with the removal of side-effects. These transformations introduce a new construct, the generalized-side-effect (gse). It is basically a triple. The first element is a list of statements to be executed before evaluation of the second element. The second element is an expression which is the "value" of the gse. The third element is a list of statements to executed following the evaluation of the second element.

```
B1: ?A = gse (?B, ?C, ?D) ==> ?B; ?A = ?C; ?D;
      where: ?D doesn't-modify ?A

B2: ?A++ ==> gse ((), ?A, (?A = ?A + ?B))
      where: ?A has-increment ?B

B3: ?A ==> gse ((), ?A, ())

B4: (gse (?A, ?B, ?C)) arith-op (gse (?D, ?E, ?F))
        ==>
      gse ((?A; ?D), (?B arith-op ?G), (?C; ?F))
     where: ?D does-not-modify ?B
            ?G is ?E with ?C propagated-thru
```

Figure 3-1, continued.

C: Transformations which determine when one construct modifies another. A crucial concept in the correct movement of side-effects.

```
C1: () does-not-modify ?A

C2: (?A = ?B; ?C) does-not-modify ?D
      where: ?A is-distinct-from ?D
             ?B does-not-modify ?D
             ?C does-not-modify ?D

C3: (?A bin_op ?B) does-not-modify ?C
      where: ?A does-not-modify ?C
             ?B does-not-modify ?C

C4: ?A does-not-modify ?C
      where: ?A is-simple-expr
              (i.e. constant or variable)
```

D: Transformations which determine when one expression is guaranteed to be distinct from another (i.e. one cannot alias the other).

```
D1: ?A is-distinct-from ?B
      where: ?A is-simple-expr
             ?B is-simple-expr
             ?A is-not-equal ?B

D2: ?A[?B] is-distinct-from ?C
      where: ?C is-simple-expr
             ?A is-distinct-from ?C

D3: ?A[?B] is-distinct-from ?C[?D]
      where: ?A is-distinct-from ?C
```

Figure 3-1, concluded

E: Transformations which propagate the effects of moving statements past expressions, or other statements.

```
E1: ?C is ?C with ?A propagated-thru
      where: ?A does-not-modify ?C
             ?C does-not-modify ?A

E2: ?B is ?A with (?A = ?B) propagated-thru
      where: ?B does-not-modify ?A

E3: (?D bin-op ?E) is (?B bin-op ?C)
       with ?A propagated-thru
      where: ?D is ?B with ?A propagated-thru
             ?E is ?C with ?A propagated-thru

E4: ?D is ?C with (?A; ?B) propagated-thru
      where: ?E is ?C with ?B propagated-thru
             ?D is ?E with ?A propagated-thru
```

F: Transformations which determine the size of the increment or decrement caused by an RPL operator. This is dependent on the type of the operand.

```
F1: ?A has-increment 1
      where: ?A has-data-type ?B
             ?B is-not "pointer to *"

F2: ?A has-increment 2
      where: ?A has-data-type "pointer to integer"

F3: ?A has-increment 4
      where: ?A has-data-type "pointer to pointer"

F4: ?A has-increment SIZEOF (?C)
      where: ?A has-data-type ?B
             ?B is-pointer-to ?C
             ?C is-structure
```

This can be achieved by the application of transformations B2 and B3 (the transformations most specific to the two operands).

At this point the situation is:

```
A = gse ((), I, (I = I + 1)) + gse ((), I, ());
```

In order to execute transformation B4, it must first be determined that:

```
        () does-not-modify I
and     ?G is I with (I = I + 1) propagated-thru
```

The first clause is satisfied by C1 (nil doesn't modify anything). E2 will satisfy the second clause provided that (I + 1) does-not-modify I. It doesn't (C3 plus C4 and C4). So ?G is bound to (I + 1) and we have:

```
A = gse ((), (I + (I + 1)), (I = I + 1));
```

Now B1, the original transformation can be applied, as long as (I = I + 1) doesn't modify A. It doesn't (it takes C1-C4 to prove it). Finally, executing B1, the original goal is satisfied:

```
A = I + I + 1;
I = I + 1;
```

Suppose that the original statement had been:

```
A = I++ + *J;
```

where the second operand is an arbitrary reference to memory. The series of transformations, outlined above, would have failed in trying to satisfy:

```
?G is *J with (I = I + 1) propagated-thru
```

Since I does not match *J nor is it distinct-from *J (they *might* reference the same location). The PTM would then have backtracked and, in the actual system, applied:

```
?A++ ==> gse ((?A = ?A + ?B), (?A - ?B), ())
  where: ?A has-increment ?B
```

Eventually satisfying the initial goal with:

```
I = I + 1;
A = I - 1 + *J;
```

If the original statement had been:

```
I = *J + I++ + *J;
```

then no series of transformations would have been successful and the PTM would make no changes. The statement would then fall through to be translated by macro-expansion using @AINC16.

The transformation of loops is handled in a similar manner. When an RPL loop is encountered a goal of changing that loop into a PL.8 loop type is established.

The main transformation which turns RPL loops into PL.8 loops is:

```
?A ==> DO ?B = ?C TO ?D BY ?E; ?F; END DO;
  where: ?A is a loop
         ?A has loop-variable ?B
         ?A has loop-init-val ?C
         ?A has loop-bound ?D
         ?A has loop-increment ?E
         ?A has loop-body  ?F
         ?F doesnt-modify ?C
         ?F doesnt-modify ?E
```

In turn there are a number of ways to identify a loop variable, increment, or bound. These are all expressed by additional transformations.

Chapter Four

Related Work

4.1 Other Transformation Systems

A significant amount of work has been done concerning code transformation. Typical transformation systems operate on a parse tree representation of the code in question. Transformations are carried out by pattern matching the code against a stored set of patterns.

One approach is to use a large set of very specific patterns [12, 8]. By making each transform very specific, interaction between transforms is minimized. This reduces the problem of ordering the application of transforms, but limits the performance of the system to transforming only those patterns which are precisely described.

A more common approach is to use a set of small general transforms which are repeatedly applied [1, 9]. The power of such systems comes from the interaction of the transforms. Particular attention must be payed to the order in which the transforms are applied if desirable results are achieved.

The simplest approach to transform selection is through a priori ordering. Small sets of transforms can be applied in a pre-determined order to successively transform a program from one form into another [3].

Transformation systems can operate interactively, asking the user to guide the selection of transforms [8, 1, 9]. This is a useful feature for synthesis systems, insuring that the user's intentions are accurately captured, and allowing him/her to influence implementation decisions.

Systems which do not require user intervention rely primarily on a built in bias in uni-directional transformation rules to achieve the desired result [5]. Systems can also rely on meta-patterns, scoping, and explicit ordering to guide the transformation process [4]. Perhaps the most sophisticated scheme is to generate several alternatives and select between them based on some specific measure (e.g., of efficiency) [14, 2, 7].

Transforms provide a convenient formalism for expressing desirable changes. Transforms alone, however, exhibit a number of short comings. They do not inherently capture the underlying properties of the parse tree nodes. There are classes of nodes which exhibit certain properties. Certain operators exhibit commutativity, or associativity. There are symmetries, asymmetries, and hierarchies which are inherent in the underlying language.

There are many ways one might try to capture such fundamental relationships. One system uses a pattern matching algorithm with implicit commutativity and associativity [5]. Another system divided its rule base into several sections, separating transformations from rules expressing fundamental properties [1]. Both systems assumed the absence of side effects within expressions.

Primarily, the use of transformation systems has concentrated on program synthesis, and optimization. Producing implementations from high level specifications, or producing efficient code from inefficient. Up to this point, little work (if any) has been done on the use of program transformation for improving the quality of translations.

4.2 Other Approaches to Translation

Although better than a pure macro-expansion approach, the transformational approach suffers from its relatively local view of the program under consideration. It lacks any understanding of the global function of a program, or the abstraction concepts represented. A representation which made sense in one language, might not be the best choice in another language. While transformation systems can improve the quality of translation they cannot achieve the goal of producing the code that a good human programmer, programming in the target language, would produce.

Program transformation is not the only approach to high quality translation. The programmer's apprentice project at MIT is approaching the problem of translation through abstraction and reimplementation [13]. The key step in this process is obtaining an abstract description of the program to be translated. Based on that abstract description, the program can be re-implemented in the target language. If the level of abstraction obtained is high enough, the translation is freed of the idiosyncracies of the source implementation. For instance, if the original implementation used an array to implement a queue, the abstract description should recognize that a queue is being used. This would allow the new implementation to use a linked list instead.

The feasibility of such an approach has been demonstrated in the translation of a subset of Cobol into Hibol (a high level business oriented language) [6] and by the implementation of a general purpose program recognition module which can perform the required abstraction [15].

References

1. Arsac, Jacques J. "Syntactic Source to Source Transformations and Program Manipulations". *Communications of the ACM 22*, 1 (Jan. 1979).

2. Barstow, David R. An Experiment in Knowledge-Based Automatic Programming. In *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann Publishers, Inc., 1986.

3. Boyle, J.M., and M.N. Muralidharan. "Program Reusabilbity through Program Transformation". *IEEE Transactions on Software Engineering 10*, 5 (Sept. 1984).

4. Cheatham, Thomas E, Jr., Glenn H. Holloway, and Judy A. Townley. Program Refinement by Transformation. TR-10-80, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, June, 1980.

5. Darlington, J. Program Transformation and Synthesis: Present Capabilities. 77/43, Imperial College of Science and Technology, Computing and Control Department, Sept, 1977.

6. Faust, G.G. SemiAutomatic Translation of Cobol into Hibol. Master Th., Massachusetts Institute of Technology,1981.

7. Kant, Elaine. On the Efficient Synthesis of Efficient Programs. In *Readings in Artificial Intelligence and Software Engineering*,
Morgan Kaufmann Publishers, Inc., 1986.

8. Kibler, D.F., J.M. Neighbors, and T.A. Standish. Program Manipulation Via an Efficient Production System. Proceedings of the Symposium on AI Programming Languages, Aug., 1977.

9. Kott, Laurent. Unfold/Fold Program Transformations. Rapports de Recherche 173, Institut National de Recherche en Informatique et en Automatique, June, 1982.

10. Ng, David M. *RPL User's Manual.* ROLM Corporation, 1982.

11. *PL.8 Language Reference Manual.* International Business Machines, 1985. Internal Document.

12. Standish, T.A., D.C. Harriman, D.F. Kibler, and J.M. Neighbors. The Irvine Program Transformation Catalogue. Department of Information and Computer Science, University of California at Irvine, Jan., 1976.

13. Richard C. Waters. Program Translation via Abstraction and Reimplementation. A.I. Memo 949, Massachusetts Institute of Technology, Dec., 1986.

14. Wegbreit, Ben. Goal-Directed Program Transformation. CSL-78-8, Xerox Palo Alto Research Center, Sept, 1975.

15. Linda M. Wills. Automated Program Recognition. AI-TR-904, Massachusetts Institute of Technology, Feb., 1987. (MS Thesis).

CS-TR Scanning Project
Document Control Form

Date : 5/26/95

Report # AIM-962

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 36(40-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP (1) UNNUMBERED TITLE PAGE	
(2-36) PAGES NUMBERED 1-35	
(37) SCANCONTROL	
(38-40) TRGTS (3)	

Scanning Agent Signoff:

Date Received: 5/26/95 Date Scanned: 6/7/95 Date Returned: 6/8/95

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 6/8/1995

M.I.T. Libraries
Document Services