This work, in full detail, can be found in: CUCS-247-86
Lecture Notes in Computer Science, Volume #302
Springer-Verlag, 1988
"Attribute Grammar Inversion and Source-to-source translation"

Translating Between Programming Languages Using A Canonical Representation And Attribute Grammar Inversion

Rodney Farrow and Daniel Yellin
Columbia University, Dept. of Computer Science
NY, NY 10027

Extended Abstract
1987

Automatic translation between programming languages is an important tool for increasing program reusability. Often the need arises to transport a large software system from one source language environment to another. Performing such a translation by hand is a large undertaking, costly in manpower and very error-prone. For this reason, several researchers have built automated tools to aid them in particular such projects [3, 1].

In this paper we present a new methodology for building source-to-source translators. This methodology involves designing a canonical form to represent programs of all source languages involved, and using attribute grammars (AGs) and automatic AG-inversion to build bi-directional translators between the various source languages and the canonical form. To test the feasibility of these ideas, we have created a system to translate between the C and Pascal programming languages.

The basic idea behind using AG inversion to translate between programming languages is illustrated by the diagram of figure 1. In this example we would like to translate between the four programming languages, A, B, C, and D. In order to do so, we first write four invertible AGs, T_A, T_B, T_C, and T_D, specifying the translation of each language into a canonical form. We then automtically invert these specifications, obtaining the inverse AGs T_A^{-1}, T_B^{-1}, T_C^{-1}, and T_D^{-1}, specifying the translation from the canonical form back to each programming language . By composing the translators obtained by this method we are able to produce a translator between any pair of languages. For example, the translator from language A to language D can be obtained by composing the specifications T_D^{-1} and T_A. Similarly, its inverse, the translator from language D to language A, is obtained by forming the composition $T_A^{-1} \circ T_D$.

For this method to succeed, we must have a canonical form in which all source language

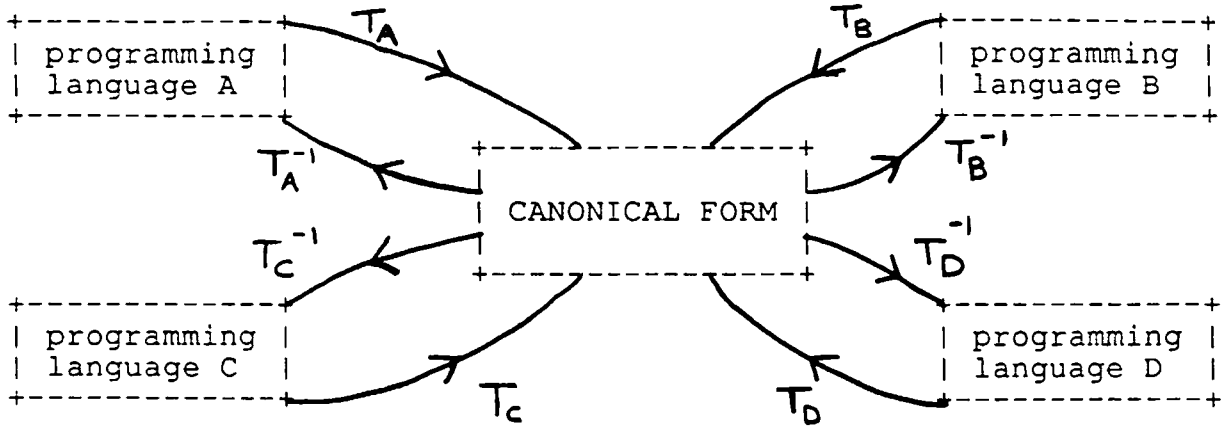

Figure 1: Using AG inversion to translate between languages

programs can be expressed. We must also be able to write invertible AGs describing the translation from the source languages to the canonical form. In the rest of this paper, we examine these issues in depth. We also describe how the method outlined above was applied to build a bi-directional translator between the Pascal and C programming languages. This was done by formulating a canonical representation in which most Pascal and C constructs can be expressed and then writing invertible AGs from the source languages into this canonical form. These AGs were automatically inverted using the INVERT system we have developed. This work has given us a better understanding of how to construct a canonical form suitable for several source languages, and of how AG inversion can be used to express complex bi-directional translations.

Our results show that AG inversion is a realistic paradigm in which to formulate the problem of translating between programming languages. It provides a useful factorization of the translation problem, helps to identify the trouble spots where the languages are incompatible, simplifies the building of complex translators, and places more of the software burden on the computer and less on the user.

1. Introduction

As described above, our method for source-to-source translations calls for defining a canonical form in which to represent all programs and for then writing invertible AGs from the source languages into the intermediate form. The success of this method hinges upon (i) our ability to find an *adequate* canonical form and (ii) our ability to write *invertible* AGs from each

programming language into this canonical form. When judging the practicality of the method, a third issue must also be considered: the efficiency of the constructed translators.

It is not hard to see that the method will work well if all the languages are closely related to one another. In such a case it is fairly obvious what the canonical form should look like, and the invertible AGs of [11] are adequate to express the translations into this canonical form. Hence this strategy can be used to build translators between dialects of a programming language or between closely related formats for representing processed manuscripts [8].

The work described in this paper shows that even when the languages are not directly related to each other, such as Pascal and C, the method described above is still a feasible approach to building source-to-source translators. We have built a pair of translators, from C to Pascal and from Pascal to C, according to this paradigm. The canonical intermediate form used by these translators is called ABSIM and is discussed later. Besides designing ABSIM, we wrote two AGs, one translating from C to ABSIM and the other translating from Pascal to ABSIM. We then used our INVERT program[1] to automatically generate the inverse AGs. All four AGs (two originals and two generated by INVERT) were then run through the Linguist [7] AG-based translator-writing-system to produce four translators: Pascal-to-ABSIM, ABSIM-to-Pascal, C-to-ABSIM, and ABSIM-to-C. The composition of appropriate pairs of these are the Pascal-to-C and C-to-Pascal translators. Example translations that these perform are listed in the appendix.

The Pascal-to-ABSIM AG is 2085 lines long, has 85 nonterminal symbols, 248 attributes, 157 productions, and 485 semantic rules. It is evaluable in 2 alternating passes. The inverse AG, automatically generated by INVERT, is 2100 lines long, has 89 nonterminal symbols, 254 attributes, 169 productions, and 521 semantic rules. It is also evaluable in 2 alternating passes.

The C-to-ABSIM AG is 2769 lines long, has 98 nonterminal symbols, 339 attributes, 175 productions, and 686 semantic rules. It is evaluable in 4 alternating passes. The inverse AG, automatically generated by INVERT, is 2873 lines long, has 98 nonterminal symbols, 321

---

[1]INVERT takes an AG in appropriately restricted form and produces another AG that describes the inverse translation.

attributes, 195 productions, and 807 semantic rules. It is also evaluable in 4 alternating passes.

In building these translators we found that it was crucial to carefully design the canonical form, ABSIM. Furthermore, since the translations from the source languages into ABSIM are fairly complex, it was also neccessary to enhance the expressiveness of invertible AGs, allowing for a more powerful paradigm than the one given in [11]. Finally, in order to achieve an acceptably efficient translator with the tools at hand we were forced to deal specially with the syntactic ambiguity that AG inversion can introduce.

Before proceeding to a discussion of these issues, we must mention that the paradigm we outline here will not perform magic; if there are constructs in language A which cannot be modeled by language B, then we cannot realistically hope to translate those portions of language A into language B (such constructs are called *non-portable* in [6]). For example, complex pointer arithmetic in C cannot be handled by our C-to-Pascal translator since there is just no good way of describing such operations in Pascal. This is not related to the inversion method of translating between languages but to the inherent difference in expressibilty between the languages.

2. Choosing the correct canonical form

Since all source language programs must be representable in the canonical form, one might think that it should contain only very low level constructs, such as assignment and goto statements, as found in many intermediate codes used in compilers. This was our opinion at the beginning of the project, however, our experience supports the opposite view.

If the canonical form is very low level, it is hard to retain program structure when translating from source to source[2]. Translating from the original program into the canonical form will essentially be the same as compiling, whereas translating from the canonical form to the target program will be similar to decompiling. In the end, there is little likeliehood that the two programs will share much in common, even though they will be semantically equivalent. By

[2]Preserving program structure is important to insure code readability, maintainability, and efficiency [6]

making the canonical form too low level, we are throwing away more information than we need to.

Instead of minimalizing the canonical representation, we want it to serve as the *greatest common denominator* between the languages (in the terminology of [6], we place a *maximality requirement* on the canonincal form). As an example, consider the for-loops of Pascal and the for-loops of C. Every for-loop in Pascal has a C for-loop counterpart but the converse is not true. Because we want to maintain program structure as much as possible, our intermediate representation takes the greatest common denominator between the two; in this case, it would include, upto syntactic isomorphism, the Pascal for-loop. Note that this will place a greater burden on our translators. Instead of blindly translating C for-loops into lower level constructs, it must now distinguish whether or not the C for-loop qualifies as a canonical form for-loop. If so, it translates it to that construct; otherwise it has no choice but to replace the for-loop by some other compatible structure (such as a while-loop).

Another reason not to make the canonical form a low level language concerns the nature of invertible AGs. Let T_A be the AG translating the language A into the canonical form and T_A^{-1} its inverse. If the canonical form is very low level, then the translation T_A will be many-to-one in the extreme. For example, if the canonical form doesn't contain any iterative loop structure but uses gotos instead, then one will not be able to tell, looking at a canonical form program containing gotos, whether the original program used gotos or for-loops. In terms of the inverse AG, this means that T_A^{-1} will have a very ambiguous context-free grammar. For a given canonical form program, there may be many parses, each producing different translations. Since each parse may be found to be syntactically or semantically invalid as more of the program is parsed and the semantic tree is evaluated, this introduces much inefficiency in the generated translators.

For our Pascal and C translators, we at first chose a widely-known intermediate representation used for compilers (a variant of Ucode) to serve as our canonical form. We soon discovered the pitfalls of this choice, as described above. Instead, we developed our own canonical form and custom designed it to reflect the greatest common denominator between Pascal and C. It omits

any idiosyncrasy peculiar to only one of the languages, while reflecting, as much as possible, the structure common to both languages. For example, Pascal and C have different conventions on returning function values. Whereas C uses a "return" statement, Pascal uses function assignment. The C convention provides an implicit transfer of control to the end of the function whereas the Pascal convention provides an implicit temporary variable. Our canonical form, being a common denominator between the languages, has neither of these capabilities. Therefore, when translating a C function into the canonical form, the implicit transfer of control of the return statement must be made explicit (using a goto). Similarly, the implicit temporary variable supplied by the function name in Pascal programs must be allocated explicity in the canonical form.

We found that using a canonical form to represent both Pascal and C programs had several benefits. First of all, it provided a well-defined factorization of the problem. Instead of translating directly between the two languages, translating into the canonical form neatly splits the problem into two subtasks. Secondly, by demanding a "greatest common divisor" canonical form, attention is focused very early in the project on identifying differences between the languages. Those incompatible areas (where there is no good way to mimic the expressiveness of one language in the other) can then be isolated. The approach to language translation advocated in [2] has many similarities with our methodology, except that we use a common canonical form instead of isomorphic *sublanguages*.

It is interesting to note that both the original and inverted AGs, translating C to and from the canonical form, requires more semantic processing than the original and inverted AGs for Pascal. Recall that the generated translators for C required 4 alternating passes, whereas those for Pascal required only 2 passes. This is because the canonical form we developed bears a closer resemblence to Pascal than to C. (There are more constructs in C that have no direct conterpart in Pascal than vice versa).

3. Writing invertible translators

After choosing an appropriate canonical form, the next step is to write invertible AGs from the source languages into this representation. In [11] it is shown how AGs can be inverted if they conform to a very restricted form. Such AGs are called *restricted inverse form grammars* (RIFs). We found the RIF restrictions to be too severe for the complex specifications needed to translate between Pascal and C.[3] For this reason we have loosened the restrictions on RIFs to obtain *generalized restricted inverse form grammars* (GRIFs). GRIFs are capable of expressing much more intricate translations, but can still be inverted to form efficient inverse specifications. Based on these ideas, we have constructed the INVERT system. It accepts a restricted AG as input and delivers the inverted AG as output, as indicated in figure 3-1.

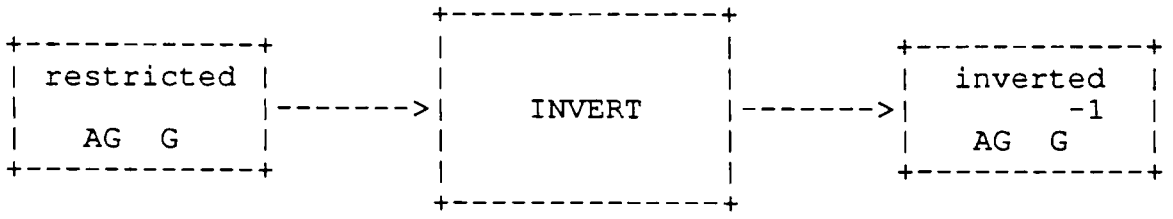

Figure 3-1: The INVERT system

The basic idea behind RIFs is to associate with each nonterminal a special attribute, called the TRANS attribute. This attribute will always express the translation of the subtree beneath any nonterminal in a semantic tree. Furthermore, this attribute must be defined by a restricted functional form. Although other attributes can be associated to nonterminals and can be computed by arbitrary semantic functions, they can only indirectly influence the translation. A RIF can be easily inverted, production by production. For details, see [11]. An example production and the inverse productions that would be generated for it by the INVERT system is given in figure 3-2. This example is a simplified version of a production found in our C-to-ABSIM AG. Since C provides implicit conversion between the integer and char data types, the two types can be intermixed. In Pascal, however, explicit conversion via the "ord" and "chr" functions must be supplied. Whereas production p will translate an expression "e" to "ord(e)", "chr(e)", or just "e", the inverse productions, pI_1, pI_2, and pI_3 will translate "ord(e)",

[3]Although one can show that *any* AG can be converted to a RIF [10], the construction produces extremely inefficient translators and cannot be used in practice.

```
p: expression ::= exp.
     expression.TRANS = if (exp.TYPE = intType and
                          expression.EXPECTED_TYPE = charType)
          then Concat['chr (', exp.TRANS, ')']
          elsif (exp.TYPE = charType and expression.EXPECTED_TYPE = intType)
          then concat['ord (', exp.TRANS, ')']
          else exp.TRANS;
... (other semantics) ...

pI1: expressionI ::= "chr" "(" expI ")".
     experssionI.TRANSINV = if NOT(expI.TYPE = intType and
               expressionI.EXPECTED_TYPE = charType)
          then ERROR else expI.TRANSINV;
... (other semantics) ...

pI2: expressionI ::= "ord" "(" expI ")".
     experssionI.TRANSINV = if NOT(expI.TYPE = charType and
               expressionI.EXPECTED_TYPE = intType)
          then ERROR else expI.TRANSINV;
... (other semantics) ...

pI3: expressionI ::= expI.
     experssionI.TRANSINV = if (expI.TYPE = intType and
               expressionI.EXPECTED_TYPE = charType) or
           (expI.TYPE = charType and expressionI.EXPECTED_TYPE = intType)
          then ERROR else expI.TRANSINV;
... (other semantics) ...
```

Figure 3-2: A production and its generated inverse productions

"chr(e)", and "e" all back to the string "e". Note that conditions are attached to the semantic functions of these productions to enforce the proper semantics.

In RIF grammars, each nonterminal has a single trans attribute and at each interior node of the parse tree, this attribute contains the translation of the subree beneath it. If the translation of a subtree can best be viewed as two or more parts that are not to be consecutive in the output string, it is often difficult to express the translation as a RIF. GRIF grammars allow a nonterminal to have several trans attributes, thereby allowing them to express these sorts of translations quite easily. The inverse GRIF grammar will contain one nonterminal for each trans attribute of a symbol.

Figure 3-3 gives an grammar fragment written as a GRIF. This example translates from a Pascal-like language, requiring function headings of the form: "function-name (parameter-declarations): type;" to a C-like language requiring headings of the form: "type function-name (

parameter-names) parameter-declarations". In particular, it would translate strings of the form: "f(a: integer; b, c: real):integer" to strings of the form: "int f(a, b, c) int a; real b, c;". The reason why this is difficult to express as a RIF is because, for each parameter declaration, two translations must be captured. The first gives the names of the identifiers, the second gives the actual declaration.

```
p1: funcDec ::= Id "(" parameters ")"  ":" type ";".
    funcDec.TRANS = Concat[ type.TRANS, Id.TRANS, '(', parameters.TRANS1, ')',
                            parameters.TRANS2];

p2: parameters0 ::= parameters1 ";" parameter.
    parameters0.TRANS1 = Concat[parameters1.TRANS1, ',', parameter.TRANS1];
    parameters0.TRANS2 = Concat[parameters1.TRANS2, parameter.TRANS2];

p3: parameters ::= parameter.
    parameters.TRANS1 = parameter.TRANS1;
    parameters.TRANS2 = parameter.TRANS2;

p4: parameter ::= identifiers ":" type.
    parameter.TRANS1 = identifiers.TRANS;
    parameter.TRANS2 = Concat[type.TRANS, identifiers.TRANS, ';'];

p5: identifiers0 ::= identifiers1 "," Id.
   identifiers0.trans = Concat[identifiers1.TRANS, ',', Id.TRANS];
```

Figure 3-3: A production using multiple TRANS attributes

If R is a RIF, R^{-1} its inverse, and T a tree in R translating s to m, then in R^{-1} there will exist a tree T^{-1} isomorphic to T translating m to s. Once we discover the isomorphic tree T^{-1}, it is easy to recover the string s since, in essence, we have the parse tree for s. In GRIFs, however, the process is not quite so simple. If T is a parse tree for a GRIF G, translating s to m, the inverse parse tree T^{-1} in G^{-1}, will not necessarily be isomorphic to T. In particular, a subtree of T may be duplicated several times in T^{-1}, or it may be split apart and reconstructed in T^{-1}, so that T^{-1} is no longer recognizeable as an isomorphic image of T. Nonetheless, our formulation of GRIFs ensures that T^{-1} contains enough information to allow us to recover a tree isomorphic to T. This is done by a process which essentially acts as a tree transformer. It would take the parse tree T^{-1} for m and create a new tree, T', isomorphic to T. Once this tree is obtained, it is an easy task to recover the string s. The transformation algorithm is done in one pass over the tree, and is therefore quite efficient.

An example of this process is presented in the following two figures. Figure 3-4 gives part of the parse tree for the string "int f(a,b,c) int a; real b,c;" based on the (inverted) GRIF given above. After this parse is found, it would be transformed to the tree of figure 3-5. This latter tree is isomorphic to the original parse tree for "f(a: integer; b,c: real): integer;", enabling us to recover that translation.

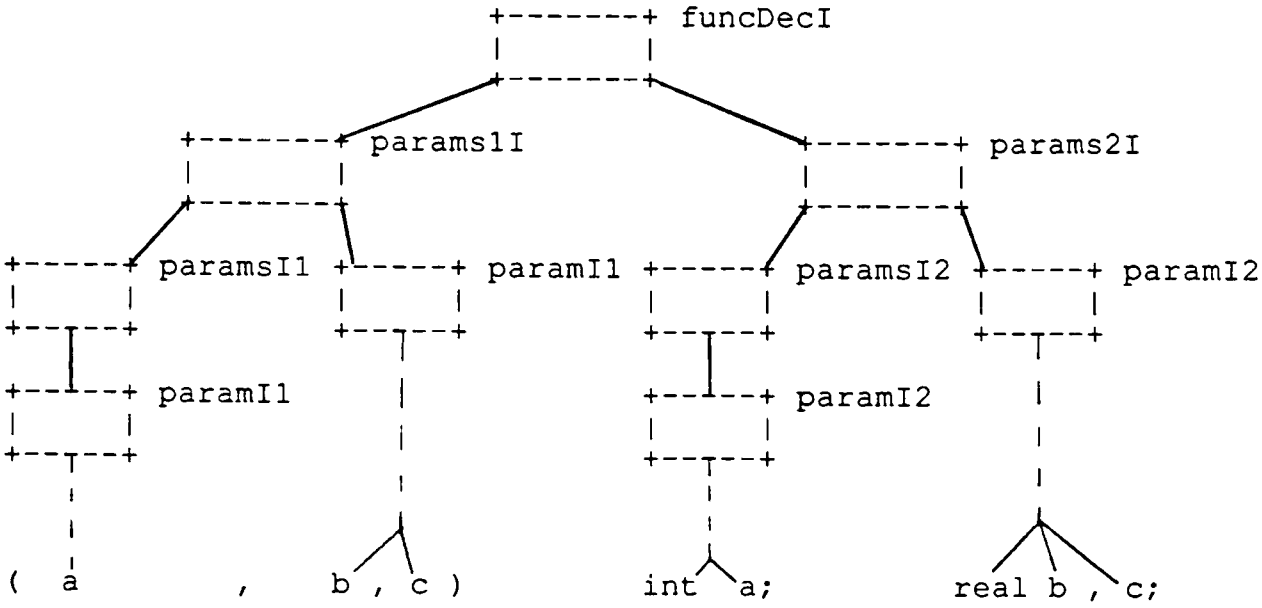

Figure 3-4: The parse tree

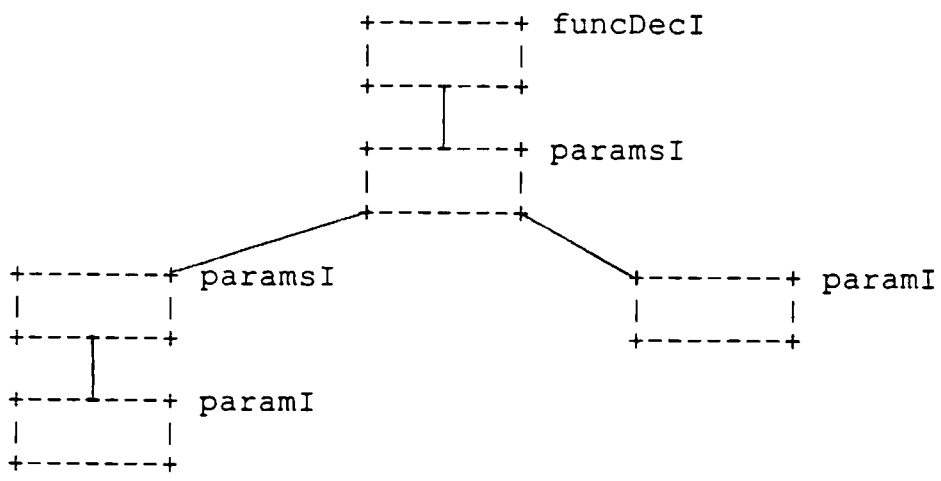

Figure 3-5: The transformed tree

It is only possible to transform an inverse tree to an isomorphic representation of the original tree if sufficient information is present in the inverse tree. Our formulation of GRIFs insures that any valid inverse tree will always contain enough information. A formal description of GRIFs,

along with the inversion and tree transformation algorithms, is given in [10]. Other extensions to RIFs, as implemented in the INVERT system, are also described there.

4. Dealing with many-to-one translations

An AG G describing a translation T is *many-to-one* if there exist strings x and y such that (x,s) $\in$ T and (y,s) $\in$ T. Similarly, if there exists a string s such that (s,x) $\in$ T and (s,y) $\in$ T then G is said to be *one-to-many*. In such a case G will specify two unique parse trees for s, one translating it to x, the other to y. If G is many-to-one then its inverse G^{-1} will be one-to-many.

The method we have described for translating between programming languages often results in one-to-many inverse AGs. For example, the C strings "X = X + 1", "X += 1", and "X++" will all be translated to the Pascal string "X := X + 1". The inverse translator therefore will specify that "X := X + 1" can be translated to any one of the above strings. Unfortunately, the ambiguity in the genenerated inverse AG can create problems for our translators if we rely on a typical deterministic shift/reduce parser. In such a case we have no method for analyzing multiple parses, but arbitrarily choose one parse. If this parse is later invalidated due to as of yet unseen syntax or as of yet uncomputed semantics, we have no method for backtracking[4].

A general solution for solving this problem would be to build a system for evaluating ambiguous AGs. Such a system would allow multiple parses for a given input to be maintained (for example, by using Earley's algorithm for finding all parses for an ambiguous context-free grammar). It would throw away a parse if it (i) determines that the parse is syntactically or semantically invalid, or (ii) determines that a "better" parse exists. Useful metrics for evaluating how "good" a parse is might be based on the length or amount of *structure* of the code generated for the parse.

Unfortunately, our current AG evaluator interfaces with a standard shift/reduce parser (YACC) and assumes that only one parse exists. Part of our ambiguity problem was solved by *collapsing productions* [11] to statically remove ambiguity from the grammar. Although this can only solve

---

[4]Even if an AG G describes a one-to-one translation and is unambiguous, it is possible that the generated inverse G^{-1} will be ambiguous. Hence we need to prepared to handle ambiguity even for one-to-one mappings.

the problem in limited circumstances, it was quite useful in practice. In the INVERT-generated ABSIM-to-C AG, 16 out of 206 productions were collapsed. The INVERT system also allows the GRIF writer to specify that a production (or part of a production) is not to be inverted, thereby allowing one to remove productions causing ambiguity from the inverse grammar.

5. Conclusions

In this paper we have presented a new methodology for building source-to-source translators and described a prototype system that we have implemented for translating between the languages C and Pascal. Our research has identified the importance of designing a *greatest common denominator* intermediate language and has extended the capabilities of automatic attribute grammar inversion beyond those originally proposed in [11]. By actually writing and inverting two large grammars (using the INVERT system), we have demonstrated the feasibilty of this approach.

Our experience has also helped us understand just where in the system the real work is being done. Source-to-source translation is, in many respects, just an extended series of pattern-matching and replacement, and that is what our translators do. The patterns to be found in the source string are described by the syntax and (some of the) semantics of the AG that translates from source to canonical intermediate form. Furthermore, these are the same source string patterns that will be generated in the output of the intermediate form to source translation. The intermediate form is essentially a catalogue of the high-level, language-independent patterns out of which all programs in any of the languages is constructed. The rest of the semantics of the AGs (those semantic rules that tell how to synthesize the TRANS attributes) describe the correspondence between the patterns of a particular language and the high-level patterns of the intermediate form.

Our research concerning a greatest common divisor canonical form is similiar, in many respects, to the work of [2]. Although there has been much research on source-to-source translations, the idea of automatically inverting a translation specification to form an inverse specification has not been widely studied. It was proposed in [9] for generalized syntax-directed translation schemata and first formulated for a restricted class of attribute grammars (RIFS) in

[11]. It has recently been suggested [8] that bi-directional translators formed from AG inversion be used in a system to support the exchange of electronic manuscripts. The basic principle behind AG-inversion, interpreting certain semantic rules of an AG as themselves being context-free rules of another AG, is similar to efforts described in [4, 5] to *compose* two AGs rather than invert one.

References

[1] G. Arango, I. Baxter, P. Freeman, C. Pidgeon.
TMM: Software Maintenance by Transformation.
IEEE Software 3(3):May, 1986.

[2] P. F. Albrecht, P. E. Garrison, S. L. Graham. R. H. Hyerle, P. Ip, and B. Krieg-Bruckner.
Source-to-Source Translation: Ada to Pascal and Pascal to Ada.
SIGPLAN Notices 15(11):183-193, 1980.

[3] James M. Boyle and Monagur N. Muralidharan.
Program Reusability through Program Transformation.
IEEE Transactions on Software Engineering SE-10(5):575-588, 1984.

[4] Harald Ganzinger and Robert Giegerich.
Attribute Coupled Grammars.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction.* ACM-SIGPLAN, June, 1984.
Published as Volume 19, Number 6, of *SIGPLAN Notices.*

[5] Robert Giegerich.
On the Relation between Descriptional Composition and Evaluation of Attribute Coupled Grammars.
Technical Report, University of Dortmund, D-4600 Dortmund 50, West Germany, February, 1986.
Preliminary version.

[6] Bernd Kreig-Bruckner.
Language Comparison and Source-To-Source Translation.
In P. Pepper (editor), *Program Transformation and Programming Environments*, pages 299 - 304. Springer-Verlag, 1984.

[7] Rodney Farrow.
User Manual for Linguist, version 3.0.
Technical Report, CS Division, EECS Dept., University of California, Berkeley, December, 1984.

[8] S. Mamrak, M. Kaelbling, C. Nicholas, and M. Share.
A Software System To Support The Exchange Of Electronic Manuscripts.
Technical Report, Department of Computer and Information Science, Ohio State University, June, 1986.
Submitted for publication.

[9] Steven P. Reiss.
Inverse Translation: The Theory of Practical Automatic Programming.
PhD thesis, Yale University, December, 1977.

[10] Daniel M. Yellin.
Source-To-Source Translations Using Automatic Attribute Grammar Inversion.
PhD thesis, Columbia University, New York, New York, 1986.
In preparation.

[11] Daniel M. Yellin and Eva-Maria M. Mueckstein.
The Automatic Inversion of Attribute Grammars.
IEEE Transactions on Software Engineering SE-12(5):590 - 599, May, 1986.

6. Appendix

Example 1: Translating from Pascal to C

The original Pascal program:

```
program swapper(input, output);
var a, b: integer;

procedure swap(var x, y :integer);
var temp: integer;
begin
     temp := x;   x := y;   y := temp
end;

procedure swapAndIncrementLarger(var s, t: integer);
begin
     swap(s,t);
     if s > t then s := s + 1  else t := t + 1
end;

begin
write('enter first integer:');  read(a);
write('enter second integer:'); read(b);
swapAndIncrementLarger(a, b);
write('first equals ', b);
write('second equals ', a)
end.
```

The translated C program:

```
#include <stdio.h>

void  swap(x, y) int *x, *y;
{     int temp;
      temp = *x; *x = *y; *y = temp;
      }

void  swapAndIncrementLarger (s,  t) int *s, *t;
{     swap (s, t);
      if (*s > *t) *s = *s + 1;
      else *t = *t + 1;
      }

int  a,  b;
main ( )
{     printf("%s", "enter first integer:");
      scanf ("%d", &a);
      printf("%s", "enter second integer:");
      scanf("%d", &b);
      swapAndIncrementLarger(&a, &b);
      printf ("%s%d", "first equals", b);
      printf("%s%d", "second equals", a);
      }
```

Example 2: Translating from Pascal to C

The original Pascal program:

```
const maxArraySize = 10;
type intArray = array [1 .. 10] of integer;
var invoice: intArray;
   i: integer;

function totalFirstN(var list:intArray; n: integer): integer;
  var i, sum: integer;
  begin
     totalFirstN := 0;
     if n <= maxArraySize
     then begin
        sum := 0;  i := 1;
        repeat
          sum := sum + list[i];
          totalFirstN := sum;
          i := i + 1
        until (i > n) or (sum< 0)
        end
  end;
```

The translated C program:

```
#include <stdio.h>

#define maxArraySize 10
typedef int intArray[10];

int totalFirstN(list, n) intArray list; int n;
{
     int i, sum;
     int tempFuncVal;
     tempFuncVal = 0;
     if (n <= maxArraySize)
     {    sum = 0 ;
          i = 1 ;
          do {
               sum = sum + list[i - 1];
               tempFuncVal = sum;
               i = i + 1;
               }
          while (! ((i > n) || (sum < 0)));
          }
     return tempFuncVal;
     }

int i ;
intArray invoice;
```

Example 3: Translating from C to Pascal:

The original C program:

```
int overTwoHundred;
int typeT(c) int c;
{
     if (c >= 'a' && c <='z' || c >= 'A' && c <= 'Z')
          return('a');
     else if (c >= '0' && c <= '9')
          return('0');
     else { overTwoHundred = overTwoHundred || (c > 200); return(c); }
}
```

The translated Pascal program:

```
#include "pasLib.i"
var overTwoHundred: integer;
function typeT(c: integer): integer;
label 1;
begin
     if ((c >= ord('a')) and (c <= ord('z'))) or
       ((c >= ord('A')) and (c <= ord('Z')))
     then
          begin
          typeT := ord('a');
          goto 1
          end
     else
     if (c >= ord('0')) and (c <= ord('9'))
     then
          begin
          typeT := ord('0');
          goto 1
          end
     else
          begin
          overTwoHundred := boolToInt( intToBool(overTwoHundred) or (c > 200));
               begin
               typeT := c;
               goto 1
               end
          end
1:    end;
```

(NOTE: pasLib.i contains some general pascal routines, such as intToBool and boolToInt, used in translated programs)

Example 4: Translating from C to Pascal

The original C program:

```
typedef int intArray[10];
void bubbleSort(S)
intArray S;
{     int beginIndex = 0, endIndex = 9, i;
      for (i = beginIndex; i <= endIndex - 1; i++)
      {       int j;
              for (j = endIndex; j >= i + 1; j--)
              {
                  if (S[j] < S[j-1] )
                  {
                        char temp;
                        temp = S[j-1];
                        S[j-1] = S[j];
                        S[j] = temp;
                        }
                  }
            }
      }
```

The translated Pascal program:

```
#include "pasLib.i"
type intArray = array [0..9] of integer;

procedure bubbleSort(S : intArray);
 label 1;
  var beginIndex, endIndex, i : integer;
     j : integer;
     temp : char;
begin
      beginIndex := 0;
      endIndex := 9;
      for i := beginIndex to endIndex - 1 do
      begin
            for j := endIndex downto i + 1 do
            begin
                  if S[j] < S[j-1] then
                  begin
                        temp := chr(S[j-1]);
                        S[j-1] := S[j];
                        S[j] := ord(temp)
                  end
            end
      end;
1: end;
```