

INTELLIGENT TRANSFORMATIONS OF FORM REPRESENTATIONS FOR PROGRAMMING LANGUAGES

Ronald B. Finkbine
CS Department
New Mexico Tech
Socorro, NM 87801
(505) 835-5209
finkbine@minos.nmt.edu

Jeffery Putnam
CS Department
New Mexico Tech
Socorro, NM 87801
(505) 835-4471
jefu@minos.nmt.edu

1 Abstract

Software re-engineering aids the programmer by attempting to add the qualities of maintain-ability and understandability to existing computer software. The practice of re-engineering has been constrained by the supposition that algorithms written in poor languages cannot automatically be engineered to an acceptable form. This paper describes a software translation system that ports existing programs into an intermediate language ALICE (Annotated Language Intermediate Code Environment) suitable for representing algorithms from most major families of computer programming languages. An expert system capable of intelligent reasoning about programs will modify and improve the code while programs are represented in the intermediate format.

2 Introduction

The intent of software re-engineering is to improve existing computer software with respect to the ability of the maintenance programmer to understand and maintain the code. Much of the software in existence today was written before the development of structured methods and software engineering practice [4]. This existing software represents quite an investment of resources and, if the program has operated without error for a period of time, has a measure of dependability that should not be thrown away lightly. Previous papers [8], [9] and [10] have discussed the methodology and cost benefits of automated software translation.

Automatic software translation has been a research topic for quite a while but has suffered severe criticism from opponents. One main criticism is that the programs so produced are unidiomatic in use of target language constructs [16], [17].

The re-engineering of computer software consist of three phases: (1) a reverse engineering of the existing code, attempting to recover the program algorithms and design from the program code, sometimes known as software archeology [5], (2) improving the recovered code while in an intermediate form, and (3) a forward engineering process, expressing the recovered and improved code in a new and better-supported environment.

Step (1) can be performed by standard compiler tools and techniques, including data flow analysis and flowcharting tools. Steps (1) and (3) are being studied together in a number of areas in current research [2], [15]. Analysis and definition of step (2) is the crux of this research.

This paper discusses a general-purpose software translation system, an informal description of ALICE syntax, and current research. ALICE is not intended to be an executable language, although that is possible. It is intended to be a language-independent representation for most families of languages that allows for translation and improvement of existing code.

3 Intelligence in Translation

Source-to-source translation for a number of languages exist—but the translated source is frequently unidiomatic and inefficient. To provide a common, simple intermediate form suitable for manipulation by expert system tools and optimizers, we have chosen to design ALICE. While the ALICE system is intended for translation among all procedural languages, in general, we will frequently use COBOL as an example because of the large amount of COBOL source code, its age, and the restrictive facilities COBOL provides for modern programming style.

The main thrust of this research is to add increased levels of intelligence to the software that translates one high-level source language to another. Current compiler technology is limited to automated tools that are able to translate statements adhering to a limited set of grammars to equivalent statements in a lower-level language (either assembly or machine-code). A number of packages are available to do language translation, some commercial and some freely available. These include translations within a family of languages [3], some for a single language [4], and some crossing language boundaries [1]. Translators that use POP-11 as a goal language also exist [14].

The C language is frequently chosen as an intermediate or target language for a number of reasons. C is sufficiently low-level to be able to express algorithms efficiently, yet high enough to be relatively easy to use as a target. Compilers are available on a wide range of machines, therefore a language translator using C as an intermediate language can produce code on almost any machine supporting a C compiler and thus, the translators need not concern themselves with details of the machine instruction set. A particularly good example is Kyoto Common Lisp (KCL), which uses C as an intermediate language for compilation. It is relatively easy and straightforward to recompile KCL programs on a new machine.

While C is a popular intermediate language and is often used as a step in the compilation process where machine-level code is desired, we feel C is not an appropriate language for an intermediate in translation between high-level languages. The translation to C entails discarding a quantity of information about the semantics of the program that could be retained. The translation process, the verification process, and high-level source reorganization could be affected by the loss of information.

It is for this reason that we are designing ALICE, a high-level intermediate form that provides facilities for algorithm description, data description, and annotation. The purpose of the annotations is to allow programs that manipulate ALICE programs to determine what kinds of transformations have already taken place and why. Thus an optimizer that moves a constant expression out of the loop will be expected to annotate this both with an explanation of why the optimization was done and what the original ALICE form was. This information can also be used by both a human reviewer and by programs that subsequently need the information as a further step in the translation process.

A number of functions can be performed to enhance the translated code while it is in an intermediate form including: variable suppression, variable reduction, scope reduction, scope expansion, code structuring, data structure replacement, and algorithm replacement.

Program simplification can be enhanced by reduction in the number of variables used. Careful analysis of the usage of variables may reveal that many are not true variables, but input/output patterns that can be replaced by library calls to a special set of routines built to replace input/output functions. An example is a COBOL edit-field variable. Numeric data is moved into an edit-field to perform certain editing functions such as leading zero suppression, leading zero nonsuppression, and floating dollar sign. In addition, dataflow analysis can reveal the presence of variables that are not used concurrently, thus one could save memory and run-time by removing one.

Some language, such as COBOL and FORTRAN, provide only limited rules for describing variable scoping. In these languages, global variables are frequently used extensively. This may be considered a security risk from a software engineering point of view. Such global variables can be removed by either of two methods, by reducing the scope of variables to encompass a minimal number of statements by distinguishing among multiple disjoint uses of the same variable name or by forcing all variables to be passed as parameters to all sub-programs. Grouped items can be converted to record definitions, reducing the number of formal parameters in any procedure definition. Certain languages, such as older versions of FORTRAN, do not allow grouped items (records) and use parallel arrays. Transformation from parallel arrays to records and from records to parallel arrays are possible by manipulating programs in the ALICE form.

COBOL programs predominantly use files and tables (arrays) as data structures. Both of these structures are linear and inefficient for most uses. Files are used for operations such as sorting and merging. Tables are used in applications that need to retrieve data in a timely manner (i.e a database application). The COBOL language was designed and many COBOL programs were written when computers had much less memory than current machines and these restrictions were reflected in the design of the language. The table data structure and functions performed on files could be replaced by other data structures designed to facilitate access such as linked lists or binary tree implementations in other languages. This would produce a more easily maintained program that is much

more efficient at run-time. It has been estimated that business applications spend 25 percent of their run-time in sorting and merging data [18]. Some of this time can be recovered if more efficient data structures are used. Recognizing and replacing data structures requires intelligent software and includes aspects of machine learning. ALICE provides a system to merge translator/compiler technology and techniques with methods of artificial intelligence.

Another aspect of this project involves intelligent algorithm replacement. COBOL programs typically contain a limited set of functions performed on the table data structure (sorting, searching, minimum, maximum, etc.) that can be described in an *IF-THEN* set of rules. If these algorithms can be identified, they can be replaced along with the data structures. This would involve the development of a general-purpose software library to support COBOL intrinsic features (i.e. unlimited size of integers and floats, ability to sort ascending or descending on any field within a record) suitable for reuse. Such a library could be language independent. Depending on the features and facilities of the target language, an object-oriented library could be utilized, allowing for the full benefits of object-oriented programming [7]. Object-oriented programming methodologies will also allow for operator overloading with either compile-time or run-time discrimination of operators.

ALICE will use operator overloading as fundamental tool in data analysis and thus can be said to be “object oriented” in that it will determine appropriate operations for a computation based upon the type of operands involved.

The use of data flow and dependency analysis on the intermediate code could allow the code to be parallelized. This would permit large or long-running COBOL applications normally requiring an expensive mainframe to be parallelization of code would promote the use of smaller, less expensive computers and could drastically reduce the need for mainframe computers in some organizations.

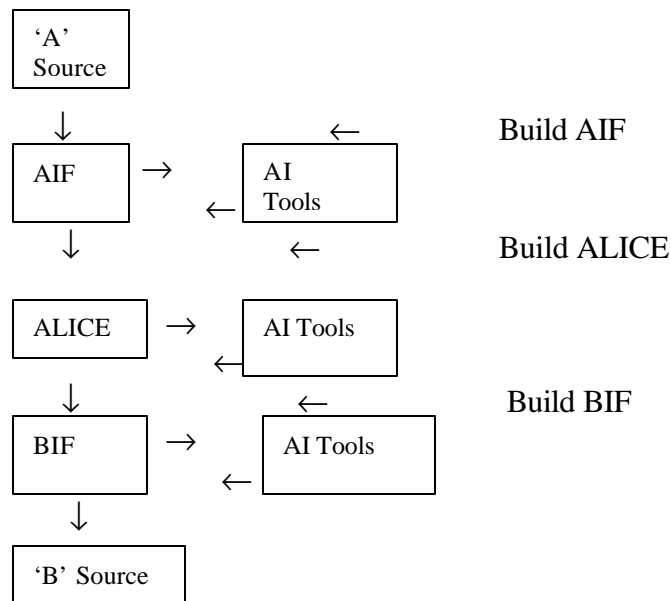


Figure 1: Translate language A to language B.

4 The ALICE Translation Process

Consider the translation from language A to language B. Language A is first translated to an annotated form AIB (language 'A' intermediate form) that is close to the structure of ALICE. In this translation, the structure of the program is not substantially changed.

Comments are included in ALICE as annotations. The translation is straightforward and a translation from this form of ALICE back to language A would be relatively simple and would, hopefully, produce code almost identical to the original source.

This stage is generally similar to standard language parsing and could use standard tools such as YACC [11]. The intermediate form is then transformed to CIF (canonical intermediate form). Annotations from the original translation are retained and annotations are added to indicate what further transformations are applied. The CIF is obtained by formal manipulations of the language intermediate form and no parser *per se* is required. Instead, it uses a series of rules to manage manipulations. These rules will be encoded in a package similar to an expert system.

The CIF may then be manipulated for source level optimization, dataflow analysis, software metrics, etc. Manipulations that result in changes in the CIF or information that may be of use in further processing may be noted in annotations. The canonical intermediate form is then translated to a language intermediate form for the target language BIF (language 'B' intermediate form). This, like the language intermediate form for language A, is close to the target language in structure, so that the translation from this intermediate form to the target language is straightforward. During the translation process further annotations may be made.

Transformation that might occur at this step include variable renaming, scope handling for variables, data structure construction or destruction, etc. The language intermediate

PERFORM A-100	loopbegin
10 TIMES	loopinit(assign(I,1))
VARYING I FROM 1 BY 1.	loopdelta(assign(i,add(i,1)))
	loopexit(greater(i,10))
A-100.	call('FIDDLE',i)
CALL 'FIDDLE' USING I.	loopend

Figure 2: COBOL fragment and ALICE equivalent

form may then be further manipulated to produce code that may be more efficient, readable and maintainable in the given target language. For example, procedures may be inlined, variable may be declared as register variables and so on.

Finally this language intermediate form is translated to the final language. This translation should be straightforward and essentially mechanical. Tools (such as graphical editors) that display the annotations (and hence the transformations) that have

been applied will be available so that a human reviewer can reject transformations that may be invalid. An expert system with learning mechanism may be useful here to assist the human reviewer in building more effective program transformation tools.

5 ALICE

An ALICE program is a textfile-based representation of an algorithm originally written in another source language that is organized linearly, explicitly identifying all critical tokens. No tokens are given special attributes due solely to their position. Tokens in an ALICE program maintain a structure of attributes describing attempted, completed and reversed transformations.

The general syntax of an ALICE statement is what appears to be a function call followed by the parameters, attributes or properties of the function. The general layout of an ALICE program is a file of procedure definitions, the first of which the main program. Each procedure is a sequential list of statements and can contain invocations of other procedures, as well as standard control sequences (*if*-statements and *loop*-statements). All statements are executable and variable definitions occur immediately prior to first-use and destroyed immediately after last-use. Execution within a procedure is sequential except for the presence of control statements and procedures can either perform a certain operations (a Pascal procedure) or return a value (a Pascal function).

Figure 2 displays a COBOL fragment and its equivalent ALICE. The first transformation rule attempted on the COBOL intermediate form representation of this COBOL fragment is the procedure-production rule. Should procedure A-100 be created from paragraph A-100? This transformation is applied if the A-100 paragraph is not used as a simple loop, but as a procedure, which can be determined by analysis of the statements within the paragraph. If procedure A-100 is created, the variable *I* is scope-reduced from a global variable to a local variable within the A-100 procedure. In this case, a procedure is not created, A-100 is recognized as a simple-loop. The variable *I* will be attributed to the procedure containing this fragment. The ALICE fragment demonstrates the general-purpose looping construct used to represent the differing varieties of loops (i.e. *repeat-until*, *do-loop*, *do-while*, and loops created from *goto*'s). The ALICE syntax, though complete, is too extensive for inclusion in this introductory paper and will be published in a subsequent paper.

6 Current Research

Currently, a number of tools are under development. A tool exists to translate COBOL-74 programs into COBOL intermediate form. A tool to translate COBOL intermediate from into ALICE is under development, as are the AI-based tools to improve ALICE representation. A tool to translate ALICE into an ADA intermediate form (possibly using DIANNA) or directly into ADA is in the planning stage.

References

- [1] Albrecht, Paul F., and Carrison, Phillip E., Graham, Susan L., Hyerle, Robert H., Ip, Patricia and Krieg-kBruckner, Bernd, *Source-to-Source Translation: Ada to Pascal and Pascal to Ada* in *Proceedings of the Symposium on the Ada Programming Language*, SIGPLAN Notices Volume 15 Number 11 (November 1980), ACM Press, 1980.
- [2] Boyle, James M. and Muralidharan, Monagur N., *Program Reusability Through Program Transformation*, IEEE Transactions on software Engineering, September 1984.
- [3] Bothe, K., Hohberg, B., Horn, Ch., and Wikarski, O., *A Portable High-Speed Pascal to C Translator* in SIGPLAN Notices Volume 24 Number 9 (September 1989).
- [4] Bush, Eric, *The Automatic Restructuring of Cobol in 1985 Software Maintenance*, IEEE Computer Society Press, 1985.
- [5] Chikofsky, Elliot j., *Case & Re-Engineering: From Archeology to Software Perestroika*, in *Proceedings of the 12th International Conference on Software Engineering*, March 26-30, 1990.
- [6] Dijkstra, E. W., *Guarded Commands, Non-Determinacy and Formal Derivation of Programs* in *Communications of the ACM*, Volume 18, Number 8 (August, 1975).
- [7] Dewhurst, Stephen C., and Stark, Kathy T., *Programming in C++*, Prentice Hall, 1989.
- [8] Finkbine, Ronald B., *Automatic Cobol-to-Ada Translation*, in *Proceedings of the First Annual Armed Force Communications and Electronics Association Midwest Regional Conference*, Dayton, OH, July 17-22, 1990 and in Technical Report 187, CS Dept., New Mexico Tech, Socorro, NM.
- [9] Finkbine, Ronald B., *An Architecture for Automatic Cobol-to-Ada Translation*, in *Proceedings of the Ninth Annual National Conference on Ada Technology*, Washington, D. C., March 4-7, 1991 and in Technical Report 188, CS Dept., New Mexico Tech, Socorro, NM.
- [10] Finkbine, Ronald B., *Artificial Intelligence Applications in Programming Language Translation* in *Thirteenth Annual Ideas in Science and Electronics Exposition and Symposium Proceedings* ISE and IEEE, Inc., Albuquerque, NM, and in Technical Report 189, CS Dept., New Mexico Tech, Socorro, NM.
- [11] Johnson, J. C., *Yacc-Yet Another Compiler Compiler in UNIX programmers Manual 2*. Revised and Expanded Version, Bell Laboratories.

- [12] Lafue, Gilles M. E., *Panel on software Re-Engineering*, in *Proceedings of the 12th International Conference on Software Engineering*, March 26-30, 1990.
- [13] Pemberton, Steven, *An Alternative Simple Language and Environment for PCs*, IEEE Software, January 1987.
- [14] *POPLOG User Guide*, University of Sussex and System Designers plc, 1987.
- [15] Sturman, J. N., *Achieving Software Reuse by Conversion and Reorganization of Software Systems*, in *proceedings of the IEEE 1990 National Aerospace and Electronics Conference, NAECON 1990*, Dayton, OH, 21-25 May 1990.
- [16] Wallis, P. J. L., *Automatic Language Conversion and Its Place in the Transition to Ada in Use: Proceedings of the ADA International Conference, 1985*, Paris, France, Cambridge University Press, 1985.
- [17] Wichmann, B. A. and Meijerink, J. G. J., *Converting to ADA Packages in Proceedings of the Third Joint Ada-Europe/AdaTEC Conference*, Brussels, Cambridge University Press, 1984.
- [18] Wolberg, J. R., *Conversion of Computer Software* Prentice-Hall, 1981.