Dartmouth College Computer Science Technical Report TR2001-395

An Implementation of Object-Oriented Program Transformation for Thought-Guided Debugging

Tiffany Wong
Dartmouth College
Department of Computer Science
June 2001

Advisor: Thomas H. Cormen

Abstract

This paper presents our design and implementation of program transformation for C++ that will be used in the context of a thought-guided debugging system. The program uses a lexical analyzer written in Flex and a grammar written in Bison that work in conjunction to scan the inputted C++ code for function definitions and class definitions. The code is then transformed to produce trace information for each defined function, while the original functionality of the code is left untouched. We also implement two additional data structures that are used for information storage during the course of the program.

1 Introduction

Despite the numerous advances that have been made in various domains of the computing field over the past several decades, most programmers still rely on very rudimentary techniques to debug their software. Many programmers are unable to utilize existing tools such as "tracers" to their full potential because the programmers lack sufficient training and experience in debugging. One possible means of implanting the programmer with the necessary skills to be effective at debugging is through the use of a thought-guided advice system [2]. This new methodology proposes to use the existing technique of algorithmic debugging [4], a process by which the debugging system acquires information about the program's expected behavior and then uses this information to localize errors, and extend the debugging technique for object-oriented programs.

There are four useful techniques that will help increase the automation of the algorithmic debugging process: test-case generation, side-effect removal, program transformation, and program slicing [2]. In this paper, we will focus on the technique of program transformation. This method consists of inserting trace-generating actions into the code in order to produce a trace file of relevant information [4] that will later be used during algorithmic debugging to help the user of the advice system locate the source of error. We present one implementation of program transformation that can be used in an advice system for C++ programmers. We will first describe an overview of the structure and design of the transformation program, and then discuss the actual implementation.

Finally, we conclude with a discussion of some of the limitations existing in this implementation of program transformation.

2 Functional Overview

In this section we describe more specifically the function of the transformation program and how it processes its inputs. We also provide some details on the internal data structures that have been implemented in order to hold information about the code while it is being processed.

2.1 Program Transformation Design

The transformation program is designed to take in as input a C++ program file and to modify it to contain numerous annotations that will be used to generate trace information about the program, while still preserving the original functionality of the code and avoiding any possible side-effects of modifying the code. An example of how the program transformation should act on a piece of C++ code is presented in Figure 1. The trace information that is returned will then be used by the advice system in order to help the user narrow down and locate the specific source of error in the C++ code. In particular, program transformation will be performed on all function definitions located in the code. The general form of a function definition is given below [6]:

```
ret_type function_name (parameter list)
{
        body of function
}
```

```
// Math function

double math(int x, double y)
{
  double z;

  x = x + 5;
  y = y * 2;

  z = x + y;

  return z;
}
```

```
// Math function

double math(int x, double y)
{
  cout << x << endl;
  cout << y << endl;

  double z;

  x = x + 5;
  y = y * 2;

  z = x + y;

  double temp = z;
  cout << temp << endl;
  return temp;
}
```

Figure 1. The fragment on the left is the original code and the fragment on the right is the code after transformation.

There are three specific regions of each function definition that must be recognized and used in the transformation process in order to produce the trace file: the function name, the parameter list, and any return statements contained in the function body.

Upon encountering a function definition in the code, the first step the program must take is to extract out the function name and the return type of the function being defined. Both of these pieces of information must be stored to an external location to be used later in the transformation process. There are no actual modifications to the code itself that need to be made at this stage.

The program must next recognize the parameter list belonging to the function. Each parameter's name and type must then be parsed out and also saved to an external location. Once the body of the function has been entered, the program should immediately insert statements that would generate the current values of all of the parameters passed to the function. This should be done before any statement in the defined function body is processed in order to avoid any changes that may be made to the values of the parameters.

Finally, the program processes the body of the function, searching for any return statements with non-void values. The program then alters the code to create a temporary variable to hold the value of the return statement, prints out its actual value, and then returns the temporary variable. The creation of this temporary variable is necessary in order to ensure that no unwanted side-effects have been generated as a result of these additional code fragments. Therefore, the original execution of the return statement should only be done once to store its value in the temporary variable. The temporary variable should then be used for all subsequent calls.

2.2 Languages and Data Structures

Our implementation of program transformation requires the use of both a grammar for C++ and a lexical analyzer in order to step through the C++ code and insert the necessary trace statements. It was also necessary to implement several data structures to store any information that needed to be maintained over the course of running the program. A symbol table is implemented in order to hold most of the state information about function

definitions. Function names and types as well as the parameter names and types for each function are all stored in the symbol table. A separate table is also maintained to hold a list of all basic and newly defined type names, in order to help the lexical analyzer to distinguish type names from other identifiers. The actual implementations of all these data structures will be discussed more fully in Section 3.3.

3 Implementation

In this section, we discuss more fully our actual implementation of program transformation. We first explain our choices for the programming languages used in the implementation and describe the algorithm we followed in the program. The section concludes with a discussion of how some of the data tables were implemented.

3.1 Language Choice

We have chosen to implement the program transformation using a lexical analyzer written in Flex and a grammar for C++ written in Bison. One of the main reasons for this decision was the availability of a pre-existing comprehensive C++ grammar [5]. Another important factor in the decision was the need for the program to recognize select pieces of the C++ language, specifically the elements making up function definitions, class declarations, and return statements. Because there are so many different ways of representing each of these elements in the code, it will be much easier to separate out the necessary pieces from the rest of the code using a grammar where tokens can be defined, rather than trying to parse the code in string format.

3.2 Program Structure

3.2.1 Creating the Flex file

The first step of any program written using Flex is to create a file for the lexical analyzer that contains the regular expressions and rules for all tokens that it needs to recognize. In our case, we want to create a rule for every valid C++ keyword, operand, and constant. In each rule, the pattern to match is simply the string itself and the action taken upon encountering each pattern will be to just return the keyword, operand, or constant. We need to maintain this list of keywords in order to help distinguish possible identifier names of functions and variables from C++ keywords when scanning the code. We now need to distinguish between type names and identifier names. We do this through the maintenance of a TypeNames table that holds a list of both basic and non-basic C++ types. All non-basic types that are defined as a `class`, `struct` or `typedef` in the code will be detected by the grammar and stored in the TypeNames table. Whenever a regular expression made up of characters, numerals, or underscores is encountered, the scanner will search for it in the TypeNames table to determine whether it is actually a type or an identifier. Finally, we add rules to recognize comments in the code so that tokens will not be analyzed unless they are part of the actual C++ code.

3.2.2 Creating the Bison grammar

The next stage of the program transformation process is to write a grammar for Bison to follow while parsing tokens. As mentioned earlier, there already exists a fairly comprehensive grammar for C++ written in Yacc [5]. Our initial thought was to take the given set of declarations and rules provided in the existing grammar and define our own

actions for each of the rules. However, the complex nature of the grammar made this task more difficult than it needed to be. Instead, we chose to write our own more simplified grammar using Roskind's work as a guide for defining our rules.

We defined a grammar to recognize the formats of function definitions, parameter lists, return statements, and class declarations. Each of these rules was broken down into its own various combinations of legal C++ structure. All other token combinations were then incorporated into a single rule that was defined as simply a continuous stream of tokens. By using some of Bison's rules of precedence [3], we were able to force the parser to differentiate the four important elements from the rest of the token stream.

3.2.3 Processing the input

The C++ program listing that needs to be transformed is given as input to the Flex program. Flex reads in the input and returns it as a series of tokens to the Bison parser. When the program first begins, there are three possibilities for each token sequence that is given to Bison. The sequence can be part of a class declaration, a function definition, or the regular token stream. Figure 2 presents a graphical depiction of how the program progresses in each of these cases.

If the parser sees a class keyword token (e.g., `class`, `struct`, `typedef`) followed by an identifier type name, then it proceeds to the rule defined for class declarations. The actions for this rule direct the program to extract out the actual identifier name from the string and store it in the TypeNames table, marking it as a non-basic type. It can now be recognized as a type name rather than an identifier in all

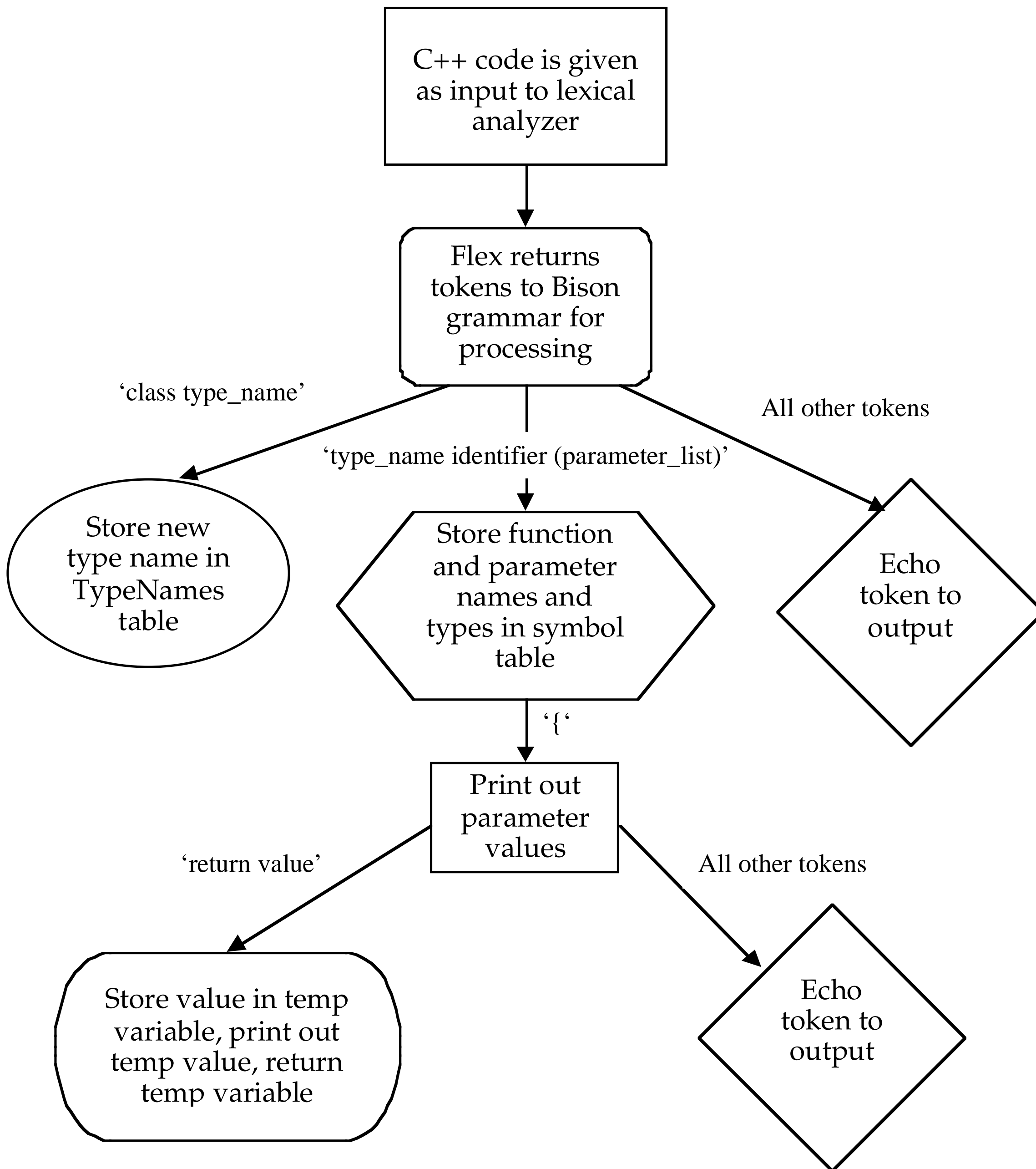

Figure 2. A graphical overview of the program transformation algorithm.

subsequent uses. After these tokens have been processed, the parser continues reading them in from the stream until it encounters another defined rule.

In the event that the parser reads in a type name followed by an identifier name, a parameter list enclosed in parentheses and a left curly brace ({), then we have the beginnings of a function definition. In this case, the first action is to store both the name of the function and its return type in the symbol table. We also store the function name in a global `char*` variable we are maintaining called *current* that will hold the name of the function we are currently processing. We will use this variable as an index to store and extract information from the symbol table at later points in the program. After the function name and type have been inserted into the table, the parameter list should then be processed. Because the parameters are comma delimited, we can extract out each parameter name and type and store it in the symbol table as well.

Once the parser has read in a { token, the body of the function starts. At this point, we want to insert trace statements that will print out the current values of the actual parameters passed into the function. We use the global *current* variable as a key to search through the symbol table so that we can retrieve the list of parameter names and types. For each parameter that is of a basic type, we can insert into the code a simple *cout* statement of the parameter name in order to print its value. If the parameter type is non-basic, we will call a specially define print function on the parameter, *parametername.debug_print()*, that will print out all the relevant values of the object. This *debug_print()* function will have already been implemented for all defined classes prior to being used in our program transformation. We have elected to create a new

function for printing purposes rather than simply overloading the `<<` operator because the programmer who defined the new class may have already overloaded the operator for his own use.

As it continues to process the remainder of the function body, Bison parses the code looking for the `return` keyword. The return statement can either be void, in which case it will simply be echoed to the output like a regular token, or it can return a value, in which case some modifications will need to be made to the code. In the latter case, the program first needs to create a temporary variable to store the return value. The type of this temporary variable will be the return type of the current function, which can be obtained from the symbol table, and will be named *temp*. Once we have inserted a statement assigning the return value to the temporary variable, we can then insert another statement to print out the value of *temp*. We will follow the same rules as described above based on whether or not *temp* is a basic type. Finally, we insert a new return statement into the code where the value being returned is now the temporary variable. By following this procedure, we can still retain all of the original functionality of the code while avoiding any possible side-effects that may have arisen had we used the actual return value instead of the temporary variable. All of these modifications are done within a new scope that we create inside the function body, in order to avoid any possible conflicts that may arise between the original code and the additions we have made.

Any other token sequence encountered while processing the function body is written out to the output file. Once the end of the function definition is encountered, the parser continues to process tokens as usual, again looking for class declarations or more

function definitions, while echoing to the output file all other token sequences in order to keep all of the original code intact.

3.3 Data Structure Implementation

In this subsection, we will discuss the implementations of the symbol table and the TypeNames table in greater detail.

3.3.1 Symbol Table

The symbol table we used is implemented as a hash table of linked lists. The structure of the symbol table is represented graphically in Figure 3. Each entry in the hash table is indexed using the function name as a key and the hash function *hashpjw* that is referenced by Aho, et al. [1]. Using this function, it is highly unlikely that any list in the table would ever grow very large. Each LinkedList structure contains a head pointer and a tail pointer for easy list traversal. All of the important information about each function is stored in the Node structures that make up the linked lists. Each Node contains four pieces of data: the function name and the function type (both stored as string pointers), a pointer to the next Node in the list, and a new linked list structure (termed a PList) containing parameter information for the function.

Like the LinkedList, each PList structure also contains head and tail pointers, as well as a private variable to hold the size of the list. This makes it simple for the program to traverse through the list of parameters when it needs to insert print statements for each one inside the function definition. Each formal parameter for a function is represented by a PNode structure in the PList. Similar to the Nodes used for function entries, the PNode

is made up of two string pointers to hold the name and type of the parameter, as well as a pointer to the next PNode in the list.

In addition to the creation of these data structures, helper functions were also written to facilitate the processes of inserting new functions into the symbol table, finding the type of a particular function, and storing function parameters into the table.

3.3.2 TypeNames Table

The second major data structure constructed for the transformation program is the TypeNames table, which holds the names of all basic types and any non-basic types that have been defined by the C++ code. The TypeNames table is internally represented as a singly linked list of Element structures. Each Element structure consists of string pointer to hold the actual name of the type, an integer called *basic* that serves to distinguish basic from non-basic types, and a pointer to the next Element in the list. When the transformation program is first started, a function is called to initialize all of the data structures before any processing of the code is done. At this point, the TypeNames table is pre-loaded with the names of all of the basic C++ types. Whenever a new type is defined in the program code, a function is called to insert the new types into the TypeNames table as a non-basic type.

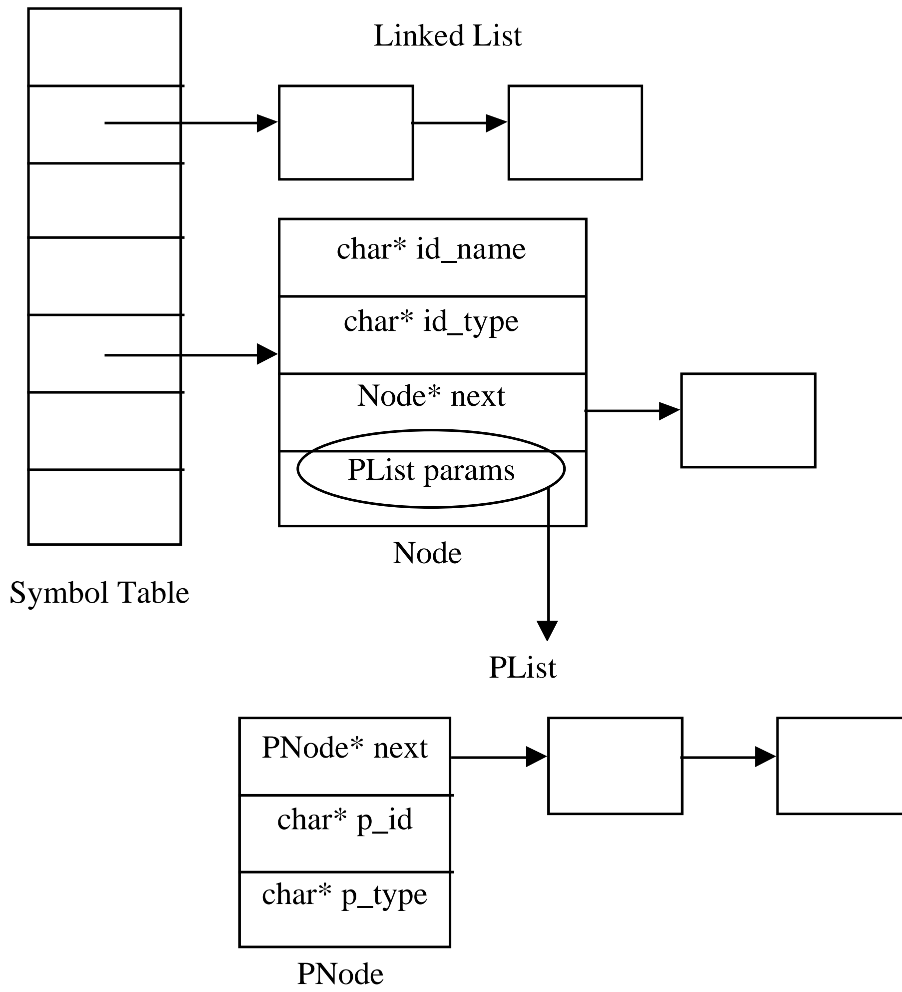

Figure 3. Structure of the symbol table.

4 Limitations

While the program transformation is comprehensive in the sense that it is able to process any given C++ code input, there are several limitations in its effectiveness as a trace generator. In particular, the program is unable to effectively handle arrays and pointers.

4.1 Arrays

When an array is given as a parameter to a function, it would be most beneficial to the user to have a trace of all the values contained in the array at that point. Unfortunately, this idea is simply impossible to implement from a programming standpoint. One problem is that there are no guarantees of knowing the size of the array since the user is never forced to pass it in as an additional parameter. With multi-dimensional arrays, we will be able to discern the sizes of all of the dimensions except for one. However, even if it were possible to obtain the size of the last dimension, there is still the possibility that the array is ragged. The last problem is that the formal parameter for the array may be declared as a pointer rather than explicitly as an array. As such, it will be impossible to make the necessary distinction to print out the array values.

As a result of these complications, we are instead forced to return much less useful information in our trace statements. The program is designed to print out merely the value of the array variable, meaning the memory address of the array's first element rather than any of the actual values stored by the array. This information will most likely not be helpful to the user but there does not seem to be any alternative at this point.

4.2 Pointers

There are also some limitations that affect our ability to properly generate trace statements for values referenced by pointers. We would ideally like to have a trace of the actual values being dereferenced by pointers. However, the problem of having null pointers and dangling references terminates this possibility. Another complication arises in the form of pointers to pointers. If we attempted to print out the value dereferenced by each separate `*`, we again run into the issue of null references. We are therefore again left with no choice but to print a trace of the pointer value, rather than the dereference value.

5 Summary

This paper presented one implementation of a program transformation for C++ that can be used in an advice system to aid programmers in debugging their software. We attempted to make the program functional for all possible inputs, but we can not be certain if the program responds correctly to all cases without further extensive testing. However, because the main goal of the advice system is to train students, rather than expert programmers, what we have implemented should be sufficient for any input they may run the program on. Although our implementation was aimed specifically to transform software written only in C++, our program uses a generic algorithm that can likely be adapted to work with other object-oriented languages.

Acknowledgments

I would like to thank my advisor, Tom Cormen, for all his help and support throughout this project. Also, Lea Wittie for all her help with debugging program, Jessica Webster for giving me the opportunity to join in with her baseball program and for answering all my Tcl questions, and John Konkle for all his help in setting up the software we needed. I would like to thank my family for all the support they have given me, both morally and financially. And finally, thanks to my friends Brad, David, Elizabeth, Rob, and Terrence for all of the great times we've had in the past four years.

References

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading: Addison-Wesley, 1986.

[2] Thomas H. Cormen and Lea Wittie. *A Methodology and System for Thought-Guided Debugging*. Unpublished manuscript, Dartmouth College.

[3] Charles Donnely and Richard Stallman. *Bison: The YACC-compatible Parser Generator*. Available at http://www.delorie.com/gnu/docs/bison/bison-toc.html. (12 February 1999, Bison Version 1.27).

[4] Peter Fritzon, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimothy. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems*, 1(4):303-322, December 1992.

[5] James A. Roskind. *LALR C++ Grammar*. Available at http://www.empathy.com/pccts/roskind.html. (22 May 1996).

[6] Herbert Schildt. *C/C++ Programmer's Reference*. Berkeley: Osborne McGraw-Hill, 1997.