Feasible C

A Writing Project Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the Requirements of the Degree

Master of Science

By

Yan Yao

Advisor: Dr. Chris Pollett

December 2004

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Chris Pollett

Dr. Agustin Araya

Dr. Kenneth Louden

Abstract

In this project, a Feasible C preprocessor is designed and implemented. The idea is to add a keyword to C called polynomial with which one can tag functions. The functions tagged with the polynomial keyword promise to run in polynomial time in their input parameters. Our translator parses our Feasible C polynomial block, and outputs standard C code. We make the assumption that it takes an OS effectively constant time to perform certain operations such as creation of local variables, and variable assignment. Similarly, we assume that arrays can be allocated from the runtime heap in time proportional to the number of elements allocated. Under these assumptions, a function verified by the Feasible C preprocessor will be polynomial time in its input arguments. Our Feasible C also can be used to simulate polynomial-time Turing Machines.

Table of Contents

List of Figures

Chapter 1. Introduction

In recent years there has been an active research area which tries to give functional characterizations of time-bounded or space bounded complexity classes. An old example of this would be Cobham's [C64] characterization of PTIME using simple base functions on the natural numbers and closure under composition and bounded recursion on notation. Unfortunately, this characterization is restricted to natural numbers and thus makes it somewhat awkward to implement ``naturally" occurring algorithms over other structures such as trees.

Another avenue of research in this area, has been to give characterizations of these complexity classes which do not explicitly mention the resource bound in question. So for example, one would try to give a functional characterization of polynomial time that does not explicitly mention polynomials anywhere. Such characterizations have been given by Bellantoni-Cook [BC92] and Leivant [L93] under the names of safe recursion and tiered recursion. Still, both safe recursion and tiered recursion have disadvantages in that many naturally occurring polynomial time algorithms do not fit into the rather rigid discipline they impose.

Recently, Hofmann [H02] gives a characterization of EXPTIME ($2^{p(n)}$, p(n) is a polynomial function of n) which allows computations over a variety of data structures such as lists or trees besides natural numbers. It also does not explicitly mention resource bounds. Thus, if a programming language were developed out of this characterization it would give the user the illusion of being able to write code of arbitrary time complexity.

Hofmann indicates that replacing general recursion with structural recursion reduces the strength of his system to polynomial time.

The purpose of my project is to come up with a subset/variant of the C language guaranteed to be polynomial time. The reason why we are choosing a variant of C is that it is a language familiar to many programmers, so the learning curve would be quicker than for an arbitrary functional language.

The idea is to add a keyword to C called polynomial with which one could tag functions with. Such a function promises to run in polynomial time in its input parameters. For code in a polynomial block, the code will be parsed by the preprocessor to make sure it satisfies the constraints of Feasible C, and, if so, the code will be converted to usual C. If the translator successfully translates such functions without errors, it would then have verified that the algorithm is in fact polynomial-time in its input parameters. We make the assumption that it takes an OS effectively constant time to perform certain operations such as creation of local variables and variable assignment. Similarly, we assume that arrays can be allocated from the runtime heap in time proportional to the number of elements allocated. The reason we say arrays can be allocated from the runtime heap is that in our Feasible C arrays are implemented as structs containing pointers. Under these assumptions, a function verified by the Feasible C preprocessor will be polynomial time in its input arguments. We also assume the lengths of arrays are all bounded by int's. That is arrays cannot be larger than 2^{32} elements. This and similar issues prevent our preprocessor from running on arbitrarily large inputs.

Lex and Yacc are used to implement this translator. Lex recognizes the expressions in the polynomial block and return them as tokens. Tokens are passed to Yacc to match into the grammars. It is a challenge to come up with a parser to pass the block of C source code. It is also more challenging to implement the constraints to our Feasible C. More importantly, besides the success of implementation of the translator, we can also theoretically prove that the polynomial function successfully passed by our translator is polynomial function.

The rest of the report is organized as follows: in Chapter 2, we introduce tools for translator design in this project. In Chapter 3, an overview of the design is given. The implementation details of the translator and the test results are provided in Chapter 4. Chapter 5 gives the proof of our theorem. Finally, conclusions are given in Chapter 6.

Chapter 2. Translator Design Tools: Lex and Yacc

A translator is designed and implemented to translate a source file which consists of "polynomial" tagged C segments. It parses the input file and checks it against a set of restrictions which are defined to guarantee the function is polynomial on it's input parameters. Lex and Yacc are two useful tools to carry out this process which is known as parsing. In other words, Lex recognizes regular expressions (tokens) while Yacc matches grammars.

2.1 Introduction to Lex

In a translator the lexer part reads the input file and recognizes tokens from the input strings. The patterns Lex uses to match the token are expressions. For every expression there is an associated action which usually returns a token to the parser.

2.1.1 Basic Specifications

The general format of a Lex document is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The rules consist of expressions and control actions. For example, the rule

```
int   printf("the string is int");
```

is to print the message "the string is int" when it finds a string "int".

2.1.2 Regular Expressions

Regular expressions are descriptions that are used in Lex to find tokens in it's input streams. The characters that form regular expressions are:

- . matches anything except the newline character.
- * matches zero or more copies of the expression. For example: [0-9]* matches zero or more copies of digits.
- [] is used to match any character in the brackets. A dash inside the brackets indicates a character range. For example: [0-9] is the expression for one digit. If the first character is a circumflex "^", it matches anything except the ones within the brackets.
- + matches one or more copies of the expression. For example: [0-9]+ matches at least one digit.
- | has the same function as logic or in any programming language. For example: yes | no matches any of those two words.
- ? matches zero or more copies of the preceding expression. For example: -?[0-9]+ matches an integer with an optional unary minus.
- () is used to group regular expressions together. For example: (ab) matches the pattern ab.

2.1.3 Lex Actions

When an expression written as above is matched, Lex executes the corresponding action. The simplest action is to ignore the input. For example:

```
[ \t]   ;
```

This action just ignores any white space.

In most cases, the action is to print out the actual text of the expression. For example:

```
[a-z]+   printf("%s", yytext);
```

will print the string in yytext . This action is so common that it may be written as ECHO:

```
[a-z]+   ECHO;
```

2.2 Introduction to Yacc

2.2.1 Basic Specifications

Every Yacc specification file consists of three sections: the definitions, (grammar) rules, and programs. The sections are separated by double percent ``%%" marks. (The percent ``%" is generally used in Yacc specifications as an escape character.) A full specification file looks like

```
definition
%%
rules
%%
programs
```

The definition section includes declaration of the tokens used in the grammar, the types of values used on the parser stack, and other odds and ends. It can include a literal block, C code enclosed in %{ %} lines. The rule section consists of a list of grammar rules. A colon is put between the left and right hand sides of a rule, and a semicolon is put at the end of each rule.

2.2.2 Yacc Grammars

The Yacc parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar. A grammar is a series of rules that the parser uses to recognize syntactically valid input. A classic example: the grammar can be used to build a calculator

statement: NAME = *expression*;

expression: NUMBER PLUS NUMBER | NUMBER MINUS NUMBER ;

where *statement* and *expression* are the nonterminal symbols and NUMBER, PLUS and MINUS are tokens returned by lexer. The vertical bar "|" means that either the proceeding rule or the following rule may have the same left-hand side (in this example which is *expression*). Following is a parse tree of the grammar to parse the input "result = 11 + 12":

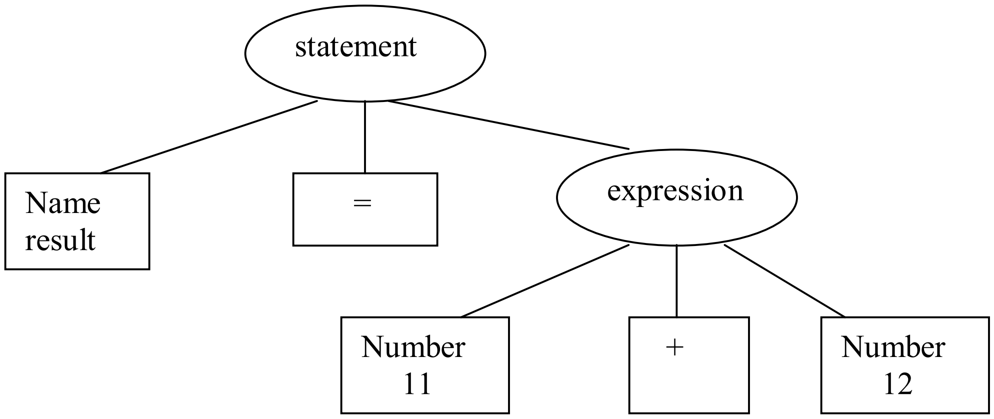

Figure 1 A parse tree

2.2.3 Recursive Rules

A rule is called recursive when its result nonterminal appears also on its right hand side. This important ability makes it possible to parse arbitrarily long input sequences.

Consider the previous example of building a calculator, the grammar can be handled in this way:

expression: NUMBER | *expression* PLUS NUMBER | *expression* MINUS NUMBER ;

The recursive rules are achieved by applying *expression* rules repeatedly. Following is the parse tree of recursive rules to parse the input " result = 10 + 2 – 3 ".

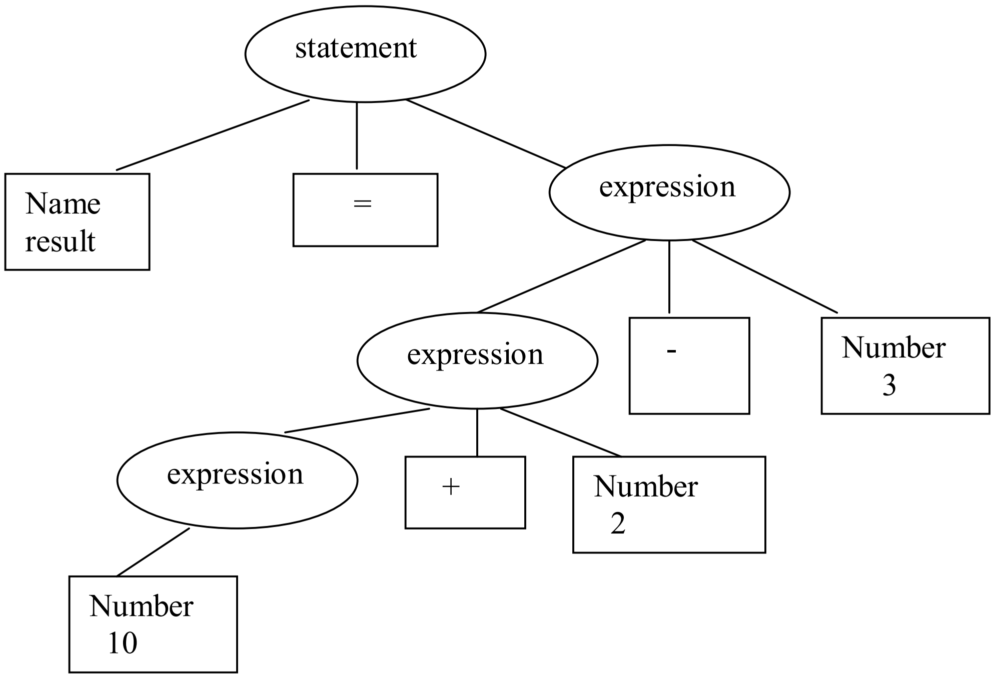

Figure 2 A recursive parse tree

Chapter 3. Overview of Design

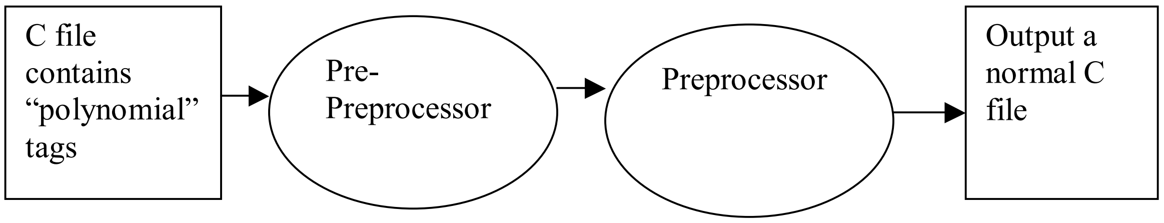

Figure 3 An overview of the Preprocessor

The translator can be divided into two parts:

The pre-preprocessor:

- Takes input file.
- If there is "polynomial tagged" #include line, it copies the content of the header file into the output file. A polynomial tagged header file can contain polynomial data types or function prototypes.
- Otherwise just copies everything into the output file.

The preprocessor

- Is fed the "polynomial" tagged C segments.
- Parses the tagged segments, and echoes the segments without tag.
- Removes the "polynomial" tag and adds necessary C segments to form a C file which can be compiled by a C compiler.

3.1 Polynomial Keyword

In my project a keyword "polynomial" is introduced. For every block of code which the programmer desires to guarantee runs in polynomial time, the programmer needs to put a keyword "Polynomial" in front of the function declaration. For example:

```
polynomial int sum ()
{
        int i;
        for (i = 0; i<10; i++)
        i+=1;
}
```

In this case function sum is a polynomial time function. When the translator sees the "polynomial" tag, it starts to parse every statement to check if the block of code is polynomial time until the end of function is met.

A polynomial tag also can be put in front of a function prototype. For example:

```
polynomial void myfunction (int i);
```

Here function myfunction is declared as a polynomial function.

Another way to use the polynomial tag is in #includes:

```
polynomial #include "datastruct.h"
```

In this case the code tagged with "polynomial" in the include header file also needs to be checked by the translator. The pre-preprocessor takes care of every appearance of a "polynomial" tagged header file in the input file. It opens the tagged header file, copies everything into a new file together with the rest of the input file to form an output file. Later on the output file will be fed to the preprocessor to check against with a set of special rules I defined for this project.

3.2 Restrictions

To check if the tagged function is a polynomial time function, the tagged functions' code needs to obey special restrictions that I define. These restrictions guarantee that a function tagged as polynomial will have a run-time that can be bounded by a polynomial in the size of its input parameters.

3.2.1 Restrictions on for loops

A *for loop* looks like this:

for (*for-initialization*; *condition*; *increment/decrement*) statement

A for loop usually consists of a loop variable, the termination condition, and the expression that updates the loop variable. The *initialization* is used to set a counter variable to its starting value. The *condition* is an expression that checks the loop variable against a termination value, and the *increment/decrement* increments/decrements the value of the loop variable. I have several restrictions to *for loops* to guarantee the loop can be performed in polynomial time.

- We disallow empty initialization statement. In C, if no initialization is needed, the initializing statement can be empty. If the condition is empty, the for-statement will loop forever unless it exits by a break, return, goto. A bad example would be:

```
for (        ;        ;        )
    {
    statement_1;
    statement_2;
    }
```

- If one of expressions is omitted (loop variable is not updated), we cannot guarantee the loop is going to terminate in a fixed amount of time. So we disallow this situation. As an example. we disallow:

```
for (i = 0; i <= 10; )
{
statement_1;
statement_2;
}
```

- If the loop variable is updated in the expression, it shouldn't be updated again in the body of the loop. An obvious mistake can be made as this example:

```
for (i = 0; i <= 10; i++)
{
        i --;
}
```

We do not allowing this situation.

- If the comparison operator in the condition is “>=” or “>”, then the loop variable should be incremented. Otherwise the loop could execute forever. For example:

```
for (i = 100; i>=10; i++)
{
        statement
}
```

The above situation is not allowed.

- In loop (for i = 0; i < j; i++), j has to be a constant or one of the function input parameters since we want to bound the runtime of the function by its input.

3.2.2 Restrictions on while loops

There are two types of *while loops*:

while (*condition*) *statement*

or

do *statement* while (*condition*)

A while-statement simply executes its controlled statement until its condition becomes false. In this case, we set a loop counter to limit the loops. Once the loop counter exceeds a certain value, the program will be terminated. To achieve this, we requires programmer to add an extra expression maxcount(constant\variable) as another while condition. The value of the variable or the constant in the parenthesis is the maximum number of times the *while* loop can be executed. When the preprocessor meets a *while* loop, it generates a *while* loop control variable name, and assigns the value of the variable or constant in the expression maxcount(constant\variable). If it is variable in the expression maxcount(constant\variable) then the variable has to be one of the function input parameters, and it also has to be an unsigned int. The control variable does a decrement for every single loop.

```
polynomial type function_name(arg1, arg2… argn)
{
        statement1;
        while((expression)&&maxcount(argi))
        {
                statement2;

        }
}
```

The output would be:

```
type function_name(arg1, arg2… argn)
{
        statement1;
        int loop_control_0 = argi;
        while((expression)&& (loop_control_0--)>0)
        {
                statement2;
        }
}
```

The preprocessor adds a statement int loop_control_0 = arg_i; right before the *while* loop statement. By doing so, it sets maximum number of loops the while loop can be run. It also adds an expression loop_control_0-->0 to decrement the loop control variable.

3.2.3 Restrictions on switch Statement

A *switch* statement is a multiple branch statement, its format looks like this:

```
        switch (expression){
         case constant1: statement;
                         break;
        case constant2: statement;
                         break;
                         .
                         .
                         .
        default: default statement
    }
```

The switch checks the value of its *expression* against the constants. For every match the statement will be executed until a *break* is encountered. The constants following the case labels must be distinct.

We do not require any special restriction on switch statements in polynomial tagged blocks.

3.2.4 Restrictions on Goto

```
    goto identifier;

        identifier: statement
```

In my project we disallowed *goto* because using goto one can define blocks with infinite loops. This would obviously result in non-polynomial time execution.

3.2.5 Restrictions on Recursive Functions

C supports recursive function calls. For every polynomial tagged function which contains a function call, a recursion control variable slot is given in the argument list by placing a $ symbol right in the front of an argument's name. If a polynomial function has a recursion control variable slot, the preprocessor will add a statement into the function to make sure the recursion will terminate. For example:

An input file might look like:

```
polynomial void myfunction (int a, int $b);
polynomial void myfunction(int i,  int $j)
{
        myfunction( i, j);
}
```

The preprocessor produces output:

```
void  myfunction(int a, int $b);
void myfunction(int i, int $j)
{
        if((j--)==0) return ;
        myfunction( i, j);
}
```

if there is a return type for myfunction, then the preprocessor adds a statement of the form return 0; to the function. For example:

```
polynomial void myfunction (int a, int $b);
polynomial int myfunction(int i,  int $j)
{
        myfunction( i, $j);
}
```

The preprocessor produces output:

```
void  myfunction(int a, int $b);
void myfunction(int i, int $j)
{
        if((j--)==0) return 0;
        myfunction( i, j);
}
```

3.2.6 Restrictions for Side-Effects

We assume several of the stdio.h functions are polynomial time. These are: printf, getchar and putchar. We allow these functions in polynomial function blocks, and also assume they are polynomial-time functions. This feature can be turned on or off by passing an optional flag "-s" which is positioned between the name of our program and the name of input file in the command line. If the flag "-s" is provided in the command line, then the preprocessor will accept those functions as polynomial function. Otherwise it will give parse error message.

3.2.7 Restrictions on Function Prototypes

If the function prototype is labeled with polynomial time, we should also check the function to make sure if it is polynomial time. If the function is not defined in the source file, an error message is given. We do not allow function to have a return type of array.

3.2.8 Restrictions on Function Definition

A function definition tagged with “polynomial” has to be checked by the translator to see if every statement is polynomial. In addition to all the restrictions I defined above, function input parameters are not allowed to appear on the left of the assignment operator.

3.2.9 Restrictions on Arrays

An array can be declared in the form of:

```
 type identifier[expression];
```

We want to know the length of an array, so we can do bounds checking. This is to prevent someone from going off the end of the array, or being able to do something pointer-like, which might lead to non-polynomial time execution. The way our preprocessor handles this restriction will be discussed in next chapter (Chapter 4).

Chapter 4. Implementation

Our translator mainly implements a C parser with some special rules. As I mentioned above, the translator has two parts: a pre-preprocessor and a preprocessor.

4.1 Define Tokens

We next discuss how C language symbols are processed. These symbols can be grouped into different categories according to their functionalities. There are reserved keywords such as if, else, and while etc. There are operators such as +, -, and *. There are strings of characters namely literals that represent numbers, strings and constants. Finally there are user defined identifiers which represent functions names variable names. In this project the lexer reads the source file, and matches the input string with expressions which represent different tokens.

Here are some expressions to match data and identifiers. To match an integer, char or float number, we use:

```
        intsuffix                       ([uU][lL]?)|([lL][uU]?)
        fracconst                       ([0-9]*\.[0-9]+)|([0-9]+\.)
        exppart                         [eE][-+]?[0-9]+
        floatsuffix                     [fFlL]
        chartext                        ([^'])|(\\.)
```

In flex, we do:

```
 [0-9]+{intsuffix}?                     {return INTEGER;}
"0"[0-7]+{intsuffix}?                   {return INTEGER;}
"0"[xX][0-9a-fA-F]+{intsuffix}?         { return INTEGER; }
{fracconst}{exppart}?{floatsuffix}?     { return FLOATING; }
[0-9]+{exppart}{floatsuffix}?           { return FLOATING; }
"'"{chartext}*"'"                       { return CHARACTER; }
```

Every occurrence of "{" indicates the beginning of a scope; and every "}" indicates the ending of a scope:

```
"{"                                 {return OPEN_SCOPE;}
"}"                                 {return CLOSE_SCOPE;}
```

For some symbols such as ";", "(",")" etc. simply return it as a text.

```
"("                              {return yytext[0];}
```

A user defined identifier is used to represent a function or variable name, and it usually starts with a letter and contains only letters and digits:

```
[a-zA-Z_][a-zA-Z_0-9]*                    { return IDENTIFIER; }
```

4.2 Pre-preprocessor

The first part of this translator we call the pre-preprocessor since it does some preparation work. It takes the input file and checks if there are any header files tagged with the "polynomial" tag. A polynomial tagged header file usually contains polynomial functions or struct/union. So the pre-preprocessor copies it into the output file which will be the input file to our preprocessor.

As an example, if the input file were the following:

```
#include <stdio.h>
polynomial #include <arraytype.h>
polynomial void array(int size)
{
        int i;
        int[] a = new int[size];
        for(i = 0; i<size; i++)
         {
                a[i] = i;

        }
}
```

and the contents of arraytype.h are:

```
polynomial typedef struct ArrayInt
           {
            int *arr;
            int len;
           };
```

after feeding the input file to the pre-preprocessor, the output file looks like this:

```
#include <stdio.h>

polynomial typedef struct ArrayInt
 {
  int *arr;
  int len;
 };

polynomial void array(int size)
{
          int i;

          int[] a = new int[size];
          for(i = 0; i<size; i++)
           {
                    a[i] = i;
          }

}
```

4.3 Preprocessor

The preprocessor is a translator consisting of a lexer and a parser. It takes the pre-preprocessor's output file as input, parses and adds necessary C segments to form a C file which can be compiled by the C compiler. Once the lexer meets a "polynomial" tag, it starts to parse. Otherwise it only echoes everything into the output file. The lexer part of the preprocessor recognizes the C keywords and returns the tokens to the parser. The parser matches the tokens to a set of rules. The rules and several C routines work together to see if the C segments can be executed in polynomial time.

4.3.1. Symbol Tables

In the preprocessor all variable and structure names and the related information such as type are stored in symbol tables. In other words a symbol table stores identifiers and its attributes. A symbol table provides the following basic operations:

- New_Table: create a new symbol table
- Delete_Table: destroy a symbol table
- Enter: put a variable into a table entry
- Lookup: find a variable in an entry

A symbol table can be implemented in different ways. The simplest way is as a linear table implemented with a fixed size array. In this case, searching of a particular symbol in a linear table takes O(n) time. Since this is only for a one scope table, it will slow the compiler to use a linear table if the number of variables in a program is very large. Compared to a linear table, hash table is a better choice for a symbol table.

4.3.1.1 Hash Table

A hash table uses a hash function to map a symbol name into a unique entry in the symbol table. It only takes O(1) time to find a symbol's entry in the hash table. Compared with the linear table, hash table offers more efficient entry access. Unfortunately, finding a perfect hash function is not easy. Sometimes different keys can be mapped on to the same entry which known as collision. A collision is resolved by placing all keys that hashed to the same entry in a linked list (chain) pointed to by this entry. As showing in

Figure 5, i and j are put into entry 2 as a linked list since the hash function mapped them into the same hash table entry.

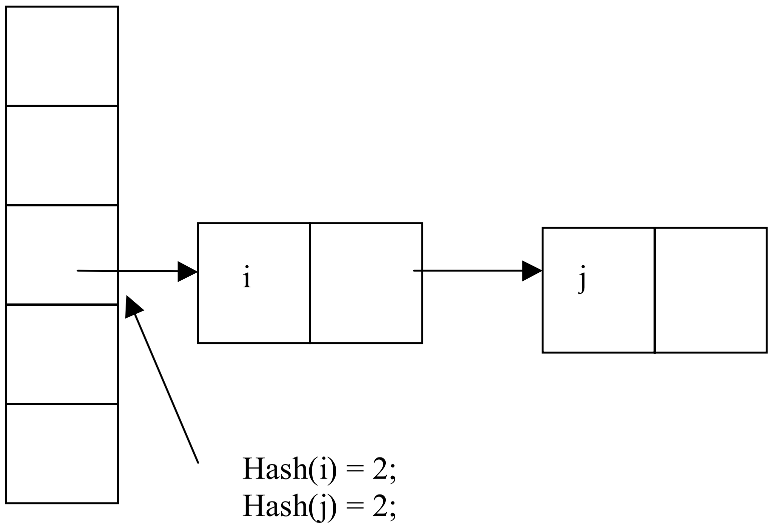

Figure 4 Collision handling in a four entry hash table

4.3.2 The Scope Problem

Like most modern programming languages, C also allows multiple scopes. The same identifier can be defined in different scopes. To solve the scope problem, this symbol table is further implemented as a stack. For every scope, it generates a new hash table (empty symbol table) and pushed it onto the stack. The innermost scope is always on the top of the stack. Once a scope is closed, we remove its hash table by popping the first item in the stack. By this method, all variables in a certain scope will be removed at once when the scope is closed. For example:

```
void example( int a, int b)
{
        int i;
        int j;
        …..
        {
                int p;
                int i;
                ….
        }
}
```

There are 3 different levels of scopes in the above example code. As shown in Figure 6, the first symbol table contains function inputs a and b; the second symbol table contains two integers i and j; and the third symbol table contains integers p and i. We can see here that i has been declared twice. The third symbol table is removed right after the first "}" is encountered, showing in Figure 7.

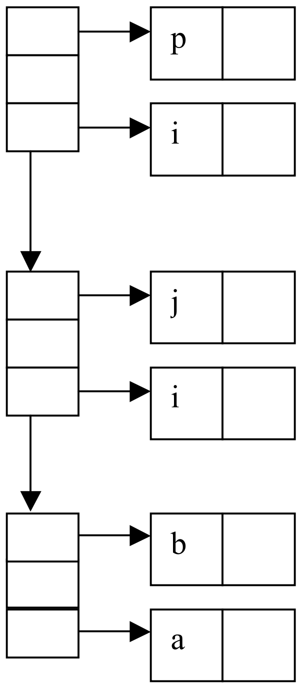

Figure 5 A three scope hash table

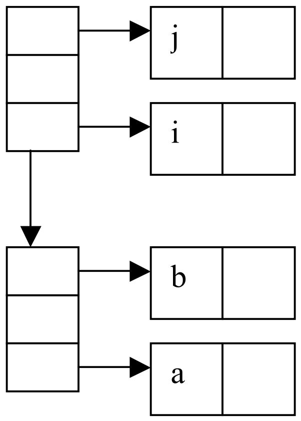

Figure 6 **A two scope hash table**

4.4 How the Restrictions are Handled

To check if the segments can be executed in polynomial time, a set of restrictions was defined as I mentioned above. The following is a description of the implementation of the restrictions.

4.4.1 Variable Declaration

An identifier can be associated with a data type through a variable declaration. The identifier is recognized by the lexer. A grammar to parse a declaration looks like this:

type identifier;

or

type identifier, identifier…;

When the parser parses a declaration, the identifier is sent to the hash function to compute the hash table entry. The entry will first be probed to see if there is already any identifier. If it is empty, the identifier and its attributes will be put into the location pointed to by the entry. Otherwise, there will be an identifier comparison to see if it's a variable re-definition or it's a collision. Collisions can be solved by using a linked list pointed to by the hash table entry. A variable re-definition will incur a parser error. For example, the input file:

```
polynomial void example (int a)
          {
               int i;
               char i;
               a+=i;
          }
```

Produces the output :

```
     void example(int a)
     {
          int i;
          char i;
parse error!
redefinition of variable name i!
```

Given a normal input file:

```
polynomial int example(int a)
{
     int i;
     i+=a;
     return i;
}
```

The output file would be:

```
int example(int a)
{
     int i;
     i+=a;
     return i;
}
```

4.4.2 Assignment

When an assignment appears in a polynomial tagged function, the preprocessor first checks the symbol table to see if the identifier on the left of the assignment operator is already stored in the table as a variable. This can be done by hashing the identifier to an entry and searching the whole list pointed by this entry. It always probes for the entry in the symbol table from the most top one to last one until a match is found. For example, given an input file:

```
polynomial int example(int a)
{
        int i;
        for(i=0; i<10; i++)
        {
                i+=a;
                 j+=i;
        }
        return i;
}
```

The preprocessor produces the output:

```
        int example(int a)
        {
                int i;
                for(i=0; i<10; i++)
                {
                        i+=a;
                         j+=
error, j is undefined variable name!
```

The next step in checking assignment is to check that function input parameters do not get assigned values in the function block. In other words all the function input parameters are not allowed to appear on the left side of a variable assignment. For example, in input file:

```
polynomial int example(int a)
{
        int i;
        for(i=0; i<10; i++)
        {
                a+=i;
         }
        return i;
```

```
                    }
```

The output:

```
                    int example(int a)
                    {
                            int i;
                            for(i=0; i<10; i++)
                            {
                                    a+=i;
              parse error, changing function input value is not allowed!
```

4.4.3 while loop

A *while* loop is an iteration of something. And it will iterate forever if the condition has not been achieved. To guarantee a *while* loop's termination, we require the programmer add an extra expression maxcount(constant\variable) as another condition. The value of the variable or the constant in the parenthesis is the maximum number of times the *while* loop can be executed. When the preprocessor meets a *while* loop, it generates a *while* loop control variable name, and assigns the value of the argument maxcount(constant\variable). If it is variable in the expression maxcount(constant\variable) then the variable has to be one of the function input parameters, and it also has to be an unsigned int. The control variable does a decrement for every single loop. To be clearer, consider a function containing while loop has the form:

```
polynomial type function_name(arg1, arg2… argn)
{
        statement1;
        statement2;
        while((expression)&&maxcount(argi))
        {
                statement3;
                statement4;

        }
```

```
                }
```

The outputs would be:

```
                type function_name(arg1, arg2… argn)
                {
                        statement1;
                        statement2;
                        int loop_control_0 = argi;
                        while((expression)&& (loop_control_0--)>0)
                        {
                                statement3;
                                statement4;

                        }
                }
```

In the example, the preprocessor generated a while loop control variable name loop_control_0, and assigned it the value of arg_i which is an unsigned int and also is one of the function input parameters. This needs to be done right before the while loop begins. The preprocessor also changes the while loop expression to:

```
while ( ( expression) && loop_control_0--) > (0)
```

By adding the expression (loop_control_0--) > 0, the *while* loop is guaranteed to be terminated after certain time. Recall that the loop control variable can only be initialized to the value of one the function input parameters or constant. The purpose of this restriction is to make sure the runtime of a polynomial function is bounded by its input. Here is an example:

The input file:

```
                polynomial int example(int a)
                {
                        int i;
                        i = 0;
                        while((i<a)&&maxcount(a))
                        {
                                i--;
```

```
          }
}
```
The output would be:

```
void example(int a)
{
          int i;
          i = 0;
          int loop_control_0 = a ;
          while ( ( i < a ) && (loop_control_0--) > 0)
          {
           i--;
          }
}
```

4.4.4 for Loop

A for loop looks like this:

for (*for-initialization*; *condition*; *increment*/*decrement*) statement

The Yacc line we wrote to parse a *for* loop:

for ‘(‘ expression; expression; expression’)’ statement

As mentioned above, every expression in the parenthesis is required to be present and needs to be checked to guarantee the for loop’s termination. Consider an example of an input where an expression is missing:

An input file:

```
polynomial void example(int a)
{
          int i;
          int j;

          for(i= 0; ; )
          {
                    j++;
                     j+=a;
          }
}
```

In this kind of situation the preprocessor simply gives a parse error message:

```
            void example(int a)
            {
                    int i;
                    int j;
                    for(i= 0;  ; )
        -
        parse error! For loop needs all three expressions!
```

A for loop can be an infinite loop even though it has all three expressions in the parenthesis. For example, if the loop control variable appears on the left of a “>” or “>=” then the loop control variable has to be decremented. Or the loop will never terminate. The following is an example:

The input file:

```
polynomial void example(int a)
{
        int i;
        int j;
        for(i= 0; i<10 ;i-- )
        {
                j+=1;
        }
}
```

The output will be:

```
                void example(int a)
                {
                        int i;
                        int j;
                        for(i= 0; i<10 ;i-- )
        parse error, bad condition!
```

Another restriction we require is that the termination value has to be set as a constant or one of the function input parameters. This is for the same reason as mentioned in the *while* loop. For example, an input file looks like:

```
polynomial void example(int a)
```

```
{
        int i;
        int j;
        for(i= 0; i<j ;i++ )
         {
                j+=1;
        }
}
```

Our preprocessor produces the output:

```
void example(int a)
{
        int i;
        int j;
        for(i= 0; i<j ;
parse error, j is not one of the input parameters!
```

4.4.5 Recursion

To deal with recursion, a recursion control variable slot represented by a $ followed by a variable name is introduced in my project. The recursion control variable slot is defined in the function prototype, and it is one of the attributes stored together with the function name in the symbol table. If there is a function call in the polynomial function, the first thing the preprocessor does is to check the symbol table to see if this function has a recursion control variable slot. The recursion control variable has to be of type unsigned int. Only polynomial functions having recursion control variables can make function calls. If the function has a recursion control variable, the preprocessor will add a line of code to control the recursion time. Let's take a look at a recursive function.

The input:

```
polynomial void myfunction (int a, int $b);
polynomial void myfunction(int i,  int $j)
{
        myfunction( i, $j);
}
```

Produces the output:

```
void  myfunction(int a, int b);
void myfunction(int i, int j)
{
        if((j--)==0) return ;
        myfunction( i, j);
}
```

In the above output file, we can see that the first line of statements: if((j--)==0) return ;.

This statement causes things to eventually terminate.

If there are several recursion control slots in a polynomial function' argument list, the preprocessor always take the last argument slot as the recursion control slot. For example, given the input:

```
polynomial void example(int a);
polynomial void myfunction(int i, int $j, int $k)
{
        example( i);
}
```

our preprocessor produces output:

```
void  example(int a);
void myfunction(int i, int j, int k)
{
        if((k--)==0) return ;
        example( i);
}
```

The preprocessor takes $k as the recursion control even though there are two recursion control variables in the argument list.

4.4.6 Function Prototype

For every function prototype with polynomial tag, the preprocessor will put it into a function list. At the same time, the attributes of this function such as function return type and the types of the input parameters are stored as well. This allows one to check in the input file:

```
void myfunction (int a, int b);
polynomial void example(int i, int j)
{
        myfunction( i,j);
}
```

that myfunction is not declared as a polynomial function. Then when it is called by function example, the preprocessor outputs:

```
void myfunction (int a, int b);
void example(int i, int j)
{
myfunction( i,j);
error, myfunction is not in the polynomial function table!
```

If the function my function is declared as a polynomial function, then the preprocessor will be happy with it. For example, given the input file:

```
polynomial void myfunction (int a, int b);
polynomial void example(int i, int j)
        {
        myfunction( i, j);
        }
```

The preprocessor produces the output:

```
void myfunction (int a, int b);
void example(int i, char j)
{
```

```
          myfunction( i,j);
}
```

Another restriction is that if there is function call/calls in this function, then there must be a '$' prefixed parameter. The purpose of this restriction is to ensure that the recursive functions calls will eventually terminate.

4.4.7 Function Calls

A function call is parsed as: function_name (arg_1, … arg_n) ;

For every function call, the preprocessor gets the type of every argument, forms a list storing type of the arguments. The arguments list is compared against the function's attributes which is stored in the polynomial function table to see if all the type of the arguments matches the type of parameters defined in the function prototype. Here is an example:

For input file:

```
polynomial void myfunction (int a, int b);
polynomial void example(int i, int j)
          {
          myfunction( i, j);
          }
```

The preprocessor finds a type mismatch in the function call:

```
void myfunction (int a, int b);
void example(int i, char j)
{
          myfunction( i,j);
parse error, type mismatch!
```

And, the arguments of a function have to be functions input parameters. For example, the input file:

```
polynomial void myfunction (int a, int b);
polynomial void example(int i, char j)
{
        int p;
        myfunction( i,p);
}
```

The output:

```
void myfunction (int a, int b);
void example(int i, char j)
{
        int p;
        myfunction( i,p
parse error, p is not one of the input parameters!
```

4.4.8 Function Definition

The rule to parse a polynomial function definition is :

"polynomial" Tag + Function Return Type + Function Name + (Parameters) +

Function Body

The function has the form as: { statements}

where statements can include the declarations, assignments, if else statements, for loops, while loops, function calls, switch statements, etc.

When a '{' is encountered, a symbol table will be generated. In this way every scope has a separate symbol table. For every appearance of '}' one symbol table is removed.

Here is a example input file:

```
polynomial void myfunction (int a, int b);
polynomial void example(int i, int j)
{
        int p;
        p = 0;
         myfunction( i, j);
        while((p<i)&&maxcount(10))
        { p++;}
        for(p=0; p<=i; p++)
        {
                p+=j;
        }

        switch(p)
        {
                case 0:
                case 1:
                default:
        }
}
```

The preprocessor checks every statement and outputs:

```
void myfunction (int a, int b);

void example(int i, int j)
{
        int p;
        p = 0;
        myfunction( i, j);
        int loop_control_0 = 10 ;
        while ( ( p < i ) && loop_control_0-- > 0)
        { p++;}
        for(p=0; p<=i; p++)
        {
                p+=j;
        }
        switch(p)
        {
                case 0:
                case 1:
                default:
        }
}
```

4.4.9 Struct and Union

Struct/union are useful data structures in C. In my project, I limited the base types of struct/union to simple data types such as int, char, double etc. Importantly pointers are not allowed. A struct is also stored in the symbol table as a data type. For example, on input:

```
polynomial struct record{
        int a;
        char b;
};
polynomial void example(int a)
{
        struct record myrecord;
}
```

The output is:

```
struct record{
        int a;
        char b;
};
void example(int a)
{
        struct record myrecord;
}
```

4.4.10 Arrays

Our preprocessor checks the bounds of arrays in a polynomial function. We have a header file arraytype.h which contains structs for int, char, float and double:

```
typedef struct ArrayChar
 {
 char *arr;
  int len;
 };
```

```
typedef struct ArrayFloat
{
 float *arr;
 int len;
};


typedef struct ArrayInt
 {
 int *arr;
 int len;
 };


typedef struct ArrayDouble
 {
 double *arr;
 int len;
 };
```

In our polynomial function, when an array declaration is met in the form:

```
int[] a =new int[expression];
```

our preprocessor maps it to:

```
ArrayInt a;
a.arr =  calloc(expression *sizeof(int));
a.len = expression;
```

In the above, *expression* must be either an input parameter or a constant. The point of these transformations is to enable the tracking of the length of array a[]. At the end of the function block the preprocessor adds a statement to free the space:

```
free(a.arr);
```

When one has a use of this array like:

a[i]

our preprocessor does the boundary checking by mapping it to:

```
a[((i<a.len&& i >= 0)? i : 0)]
```

As an example, given the input file:

```
 polynomial void array(int size)
{
        int i;

        int[] a = new int[size];

        for(i = 0; i<size; i++)
         {
                a[i] = i;
        }
}
```

the preprocessor produces the output:

```
void array(int size)
{
        int i;
        int = 0;
        ArrayInt a;
        a.arr = calloc(size, sizeof(int));
        a.len = size;
        for(i = 0; i<size; i++)
         {
                a[((i<a.len&& i >= 0)? i : 0)] = i;

        }
        free(a.arr);
}
```

Chapter 5. Proof

In order to prove that functions parsed successfully by this preprocessor are polynomial functions, operation complexity measures for C expressions and functions are introduced. We also make an assumption that it only takes OS effectively constant time to perform certain operations such as creation of local variables, variable assignment, etc. Given these assumptions and our complexity definition, we will show towards the end of this chapter that a polynomial tagged function will be polynomial time in inputs arguments. Input arguments only have finite number of bits. So it cannot necessarily run on arbitrarily large inputs.

5.1 Operation Complexity of Expressions

The operation complexities of an expression is defined as follows:

- For primary expressions, that is, variables and constants of type *int*, *long*, or *double,* the operation complexity is constant 1.
- Expressions with unary operators such as ++*expression*, *--expression,* and -*expression* have the complexity of complexity of *expression* plus 1.
- For expressions with multiplicative operators of the form of
    - *expression _1* expression_2*
    - *expression_1* / *expression_2*
    - *expression_1* % *expression_2*

  have the complexity of complexity of (*expression_1*+1) * (complexity of *expression_2*+1).
- The complexity of expressions with additive operators of the form of
    - *expression_1* + *expression_2*
    - *expresiion_1* – *expression_2*

is defined as the complexity of *expression_1* + complexity of *expression_2*.

- The expressions with relational or equality operators of the form of
    - *expression_1* < *expression_2*
    - *expression_1* > *expression_2*
    - *expression_1* <= *expression_2*
    - *expression_1* >= *expression_2*
    - *expression_1* == *expression_2*
    - *expression_1* != *expression_2*

  have the complexity of complexity of *expression_1* + complexity of *expression_2*.
- The complexity of expressions with bitwise/logical AND and OR operators of the form of
    - *expression_1* ^ *expression_2*
    - *expression_1* | *expression_2*
    - *expression_1* && *expression_2*
    - *expression_1* || *expression_2*

  is defined as complexity of *expression_1* + complexity of *expression_2.*
- For the expressions with operators such as conditional operator and assignment operators, the complexity is also defined as the sum of the complexity of every expression.
- If the expression is simply a function call, we handle it the same as the function calls. As we defined above the complexity of function call in the complexity of the function.

- If the expression is used to assign a function to a variable, we say the complexity of this expression is the complexity of the function plus 1.

5.2 Operation Complexity of Statements

We next define the complexity of a polynomial function is bounded by operation complexity for a function. The complexity of the empty function

```
type function_name (arg1, arg2… argn)
{
}
```

is zero since there are no operations in this function.

The complexity of a function in a format as following:

```
type function_name(arg1, arg2… argn)
{
        statement 1;
        statement 2;
        statement 3;
        …
        statement n
}
```

complexity = Σ complexity of *statement i* (for i = 1 to n)

The complexity for different statement types is defined as follows:

- For a statement which is a simple declaration such as

  char a, b, c;

  complexity = number of items in the list;

  This parser doesn't allow a variable to be initialized in the declaration statement. For example:

  statement int i = 0; is not allowed.

- For an assignment such as

a = *expression*;

complexity = 1 + complexity of *expression*;

- For a *return* statement

return expression

complexity = complexity of the expression;

- In a *for* loop of the form :   *for* (*i=0; i < j; i++*) statement

  complexity = j * complexity(statement(i)) ;

  or in the form:

  for (*i=10; i > j; i--*) statement

  complexity = i * complexity(statement(i)) ;

complexity(statement(i)) is the max complexity of the statement that depends on the value of i.

- For a if….else…. statement of the form

  if(*expression*) statement_1
  else statement_2

  The complexity of this statement is the greater of the complexity of statement_1 and statement_2, plus the complexity of expression of *expression*.

- For a *while* loop of the form

  while (*expression*$_1$*&&maxcount(expression*$_2$*)*) statement;

  complexity = *expression*$_2$ * complexity(statement) ;

- for a *switch* statement of the form:

```
switch (expression){
     case constant1: statement;
                     break;
     case constant2: statement;
                     break;
     ….
     case constantn: statement;
                     break;
}
```

complexity = sum of complexity of every *case* statement + complexity of *expression* ;

- For a statement block of the form:

```
{
    statement_1;
    statement_2
    …
    statement_n;
}
```

complexity of statement block = sum of the complexity of all the statements in the block.

- For a function call of the form:

  function_name(argument);

  The complexity of function call = the complexity of the function_name function

5.3 First Main Result

Theorem 1. The runtime of any C expression of the form

$$\mathrm{expr}(\mathrm{arg}_1, \ldots \mathrm{arg}_n)$$

where arg_n are the variables appearing in the expression or the runtime of any C function of the form

type_of_return_value function_name ($\mathrm{arg}_1, \mathrm{arg}_2 \ldots \mathrm{arg}_n$)

parsed by this preprocessor is bounded by

$$O((\text{abs}(\text{argument}_1)+\text{abs}(\text{argument}_2)+\ldots+\text{abs}(\text{argument}_n))^{f(m')}+f(m'))$$

$$f(m') = 2^{m'}$$

If argument_i is an array, then in the above we replace this argument by the sum of the absolute values of its elements. That is:

$$\text{abs}(\text{argument}_i\,[0])+\text{abs}(\text{argument}_i\,[1])+\ldots+\text{abs}(\text{argument}_i\,[n-1])$$

where n is the length of the array.

5.3.1 Proof by Induction

We next consider the theorem as applied to functions. In the base case Feasible C expressions are primary expressions such as :

identifier; or

constant

The complexity m' of primary expression is defined as a constant 1. The runtime of a primary expression is a computable constant therefore the theorem is true.

An expression containing two primary expressions and one operator might look like:

$$\text{expr}_1(\text{arg}_1, \text{arg}_2\ldots\ \text{arg}_m)\ \text{op}\ \text{expr}_2(\text{arg}_1, \text{arg}_2\ldots\ \text{arg}_n)$$

We will show the case where op is * as it is representative of all the other cases. By the induction hypothesis, the runtime of expr_1 is $O(\sum_{i=0}^{m} abs(\arg_i)^{f(m_1')} + f(m_1'))$. The runtime of expr_2 is $O(\sum_{j=0}^{n} abs(\arg_j)^{f(m_2')} + f(m_2'))$. For any of the C operators the time complexity of performing the operation is a most the product of its arguments, so by the induction hypothesis we have that the runtime of $\text{expr}_1(\text{arg}_1\ldots)$ * $\text{expr}_2(\text{arg}_1\ldots)$ can be bounded by $O(\sum_{i=0}^{m}(abs(\arg_i)^{f(m_1')} + f(m_1') + 1) * (\sum_{i=0}^{n}(abs(\arg_i)^{f(m_2')} + f(m_2') + 1))$.

Let $\sum_{i=0}^{m} abs(\arg_i) =$ z. Using this, the run-time of expr$_1$(arg$_1$…) * expr$_2$(arg$_1$…) is

$O(z^{f(m_1')} + f(m_1') + 1)$* $O(z^{f(m_2')} + f(m_2') + 1)$. Recall $f(m') = 2^{m'}$ So let A be

$$O((z^{2^{m_1}} + 2^{m_1'} + 1) * (z^{2^{m_2}} + 2^{m_2'} + 1)) \le O((z^{2^{m_1'}} + 2^{m_1'} + 2)(z^{2^{m_1}} + 2^{m_2'} + 2\ ))$$

$$A \le O((z^{2^{m_1'}} + 2^{m_1'+1})(z^{2^{m_1}} + 2^{m_2'+1})) = O(z^{2^{m_1'}+2^{m_2'}} + z^{2^{m_1'}} 2^{m_2'+1} + z^{2^{m_{12}'}} 2^{m_2'+1} + 2^{m_1'+m_2'+2})$$

Let z≥2(we assume the inputs to the expression is greater or equal two)

Then $A \le O(z^{2^{m_1'}+2^{m_2'}} + z^{2^{m_1'}+2^{m_2'}+1} + z^{2^{m_{12}'}+2^{m_1'}+1} + 2^{m_1'+m_2'+2}) \le O(3z^{2^{m_1'}+2^{m_2'}+1} + 2^{m_1'+m_2'+2})$

$$A \le O(3z^{2^{m_1'+m_2'+1}} + 2^{m_1'+m_2'+2})$$
$$\because 3z^{2^{m_1'+m_2'+1}} \le z^{2^{m_1'+m_2'+1}} z^2$$

$$\therefore A \le O(z^{2^{m_1'+m_2'+2}} + 2^{m_1 + m_2 + 2'}) \le O(z^{2^{(m_1'+1)*(m_2'+1)}} + 2^{(m_1 + 1)*(m_2+1)'}))$$

Let $m'' = (m_1' + 1) * (m_2' + 1), f(m'') = 2^{(m_1'+1)*(m_2'+1)}$, and the runtime of the expression is

bounded by $O(z^{2^{m_1'*m_2'}} + 2^{m_1'*m_2'})$. So induction hypothesis holds. By this kind of argument the theorem will be true for any expression.

We next consider the theorem as applied to functions. In the base case, which is a function contains zero statement in the function body:

```
type myfunction(argument1, …argumentn)
    {
    }
```

operation complexity of myfunction m'= 0;

We can see that the runtime of an empty function is bounded by a constant. Therefore the theorem is true when the function has zero statement.

Now assume any function

type function_name (argument$_1$, argument$_2$……argument$n$)

of complexity $m' < m$ has run time bounded by

$$O((\text{abs}(\text{argument}1)+\text{abs}(\text{argument}2)+\ldots+\text{abs}(\text{argument}n))^{f(m')}+f(m'))$$

$$\text{where } f(m') = 2^{m'}$$

In functions which contain more than three statements

```
myfunction (argument₁, …argumentₙ)
{
        statement_1;
        statement_2;
        statement_3;
        …..
        statement_n;
}
```

myfunction is equivalent to

```
myfunction_2(argument₁, …argumentₙ)
{
        variable declarations;
function_1(argument₁, …argumentₙ,&additional_arguments);
function_2(argument₁, …argumentₙ, &additional_arguments);
}
```

Here the first statement contains the local variable declaration statements of myfunction. They might contain statements such as `char a;', but as we have specified in our restrictions they cannot contain declarations and assignments such as `char a=10;'.

In our model we are assuming the operating system can create a local variable in constant time. The time required to allocate an array is in proportional to its length and this length will be one of the input parameters or a constant. As there are only a fixed number of such declarations, this portion of myfunction and hence myfunction_2 run will certainly run in time big-O of the given bound. In our set-up when a variable is declared it is also set to zero.

The additional_arguments passed to function_1 and function_2 come from these local variables of myfunction. Roughly, function_1 and function_2 will look like:

```
function_1(argument1, …argumentn, &additional_arguments)
{
        statement_1;
        statement_2;
        ……..
        statement_n-3;
}

function_2(argument1, …argumentn, &additional_arguments)
{
        statement_n-2;
        statement_n-1;
        statement_n;
}
```

The reason we said roughly is since some of the statement_1 … statement_n might have been variable declarations we might have less than n statements now. The complexity of the function_1 and function_2 are:

Complexity of function_1:

m_1'= complexity (statement_1) + complexity(statement_2) +……+ complexity(statement_n-3);

Complexity of function_2:

m_2'= complexity(statement_n-2) + complexity(statement_n-1) + complexity(statement_n)

So the complexity of function_1 is $m_1' <$ complexity of myfunction , and the complexity of function_2 is $m_2' <$ complexity of myfunction. If there is at least one additional argument then there had to be one variable declaration. This make the complexity of myfunction m'≥ $m_1' + m_2' + 1$ Thus, the induction hypothesis can be applied. By the induction hypothesis, the runtime of function_1 is $O(\sum_{i=0}^{n}(abs(\arg_i)^{f(m_1')} + f(m_1'))$. Note at this point all the additional arguments to function 1 are zero and so do not contribute. The complexity of additional arguments may add to function_2 is at most the runtime of function_1. So the runtime of function_2 is

$O(\sum_{i=0}^{n}(abs(\arg_i) + \sum_{i=0}^{n} abs(\arg_i)^{f(m_1')} + f(m_1')^{f(m_1')})^{f(m_2')} + f(m_2'))$. $\arg_i$ represents the sum of the arguments.

Let g represents $\sum_{i=0}^{n} abs(\arg_i)$, the runtime of myfunction is

A = $O(g^{2^{m1'}} + 2^{m_1'} + (g + g^{2^{m1'}} + 2^{m_1'})^{2^{m2}} + 2^{m2'})$

The value of g is greater or equal 2 on all but finitely many inputs (namely, the input 1). So we have

$g + 2^{m_1'} \le (2g)^{m_1'} \le g^{2m_1'}$. This gives

$g + g^{2^{m1'}} + 2^{m_1'} \le g^{2^{m_1'}} + g^{2m_1'} \le 2g^{2^{m_1'}} \le g^{2^{m_1'}+1}$

$2^{m_1'} + 2^{m2'} \le 2^{m_1' + m_2'}$

So

$A \le O(g^{2^{m1'}} + (g^{2^{m_1'}+1})^{2^{m_2'}} + 2^{m_1' + m_2'}) \le O(g^{2^{m_1'+m_2'}+2^{m_2'}} + g^{2^{m_1'}} + 2^{m_1'+m_2'}) \le O(2g^{2^{m_1'+m_2'}+2^{m_2'}} + 2^{m_1'+m_2'})$

$A \le O(g^{2^{m_1'+m_2'}+2^{m_2'}+1} + 2^{m_1'+m_2'}) \le O(g^{2^{m_1'+m_2'+1}} + 2^{m_1'+m_2'+1})$

Let $m'' = (m_1' + m_2' + 1), f(m'') = 2^{(m_1' + m_2' + 1)}$, and the runtime of the expression is bounded by $O(g^{2^{m_1' * m_2' + 1}} + 2^{m_1' * m_2' + 1})$. So induction hypothesis holds. This completes the case where the function body consists of more than three statements .

We now consider the case when the body of the function only contains one or two statements. For functions that have only one statement, the runtime is also bounded by complexity of this function. The only statement is the function can be any one of all the C statements. Here I will only give some typical examples.

```
void myfunction(argument_1, …argument_n)
{
        switch(expression)
        {
                case expression _1:statement_1
                break;
                case expression _2:statement_2;
                break;
                case expression _3: statement3;
                break;
        }
}
```

We want to show the runtime of myfunction $\leq O((\text{abs}(\text{argument}))^{f(m')} + f(m'))$

As defined in the previous chapter, the complexity of switch statement is the sum of the complexity of every case statement and complexity of expression. This gives:

Complexity of the a switch statement = complexity of every *case* statement + complexity of *expression* .

Here m_exp represents complexity of the complexity of expression, m_s1 represents the complexity of the statement_1, m_s2 represents the complexity of statement_2, and m_s3 represents the complexity of statement_3.

Complexity of m_exp m_1' < complexity of myfunction ;
Complexity of m_s1 m_2'< complexity of myfunction;
Complexity of m_s2 m_3'< complexity of myfunction;
Complexity of m_s3 m_4'< complexity of myfunction;

So the induction hypothesis can be applied.

The runtime of myfunction

$$A = O(\sum_{i=0}^{n}(abs(\arg_i) + f(m_1') + \sum_{i=0}^{n} abs(\arg_i)^{f(m_2')} + f(m_2') + \sum_{i=0}^{n} abs(\arg_i)^{f(m_3')} + f(m_3'))$$

$$+\sum_{i=0}^{n} abs(\arg_i)^{f(m_1')} + f(m_4'))$$

Let's use the g as the $\sum_{i=0}^{n} abs(\arg_i)^{f(m_1')}$

The runtime of myfunction A = $O(g^{2^{m_1'}} + 2^{m_1'} + g^{2^{m_2'}} + 2^{m_2'} + g^{2^{m_3'}} + 2^{m_3'} + g^{2^{m_4'}} + 2^{m_4'})$

Then we have $A \le O(g^{2^{m_1'}+2^{m_{12}'}+2^{m_{13}'}+2^{m_4'}} + 2^{m_1'+m_{12}'+m_{13}'+m_4'}) \le O(g^{2^{m_1'+m_2'+m_3''+m_4'}} + 2^{m_1'+m_2'+m_3''+m_4'})$

Let Let $m'' = m_1' + m_2' + m_3' + m_4', f(m'') = 2^{m_1'+m_2'+m_3'+m_4'}$, and the runtime of the function is bounded by $O(g^{2^{m_1'+m_2'+m_3'+m}} + 2^{m_1'+m_2'+m_3'+m_4'})$. So induction hypothesis holds.

The last situations we need to consider are functions consisting of two statements. The statements can be any two of the following statements: declaration, assignment, for loop, while loop, switch…. Case…., if….else…., function call. Again, I will only look at some more interesting cases. Consider, for example:

```
void myfunction(argument1, …argumentn)
{
        while(expression_1)
        {
                if(expression_2) statement_1;
                else statement_2
        }
        switch(expression)
        {
                case expression _3:statement_3;
                break;
                case expression _4:statement_4;
                break;
                case expression _5: statement5;
                break;
        }
}
```

The complexity of myfunction = complexity of while loop + complexity of switch statement;

Here m_while represents the complexity of while loop, and m_switch represents the complexity of the switch statements.

Complexity of while loop m_1' < complexity of myfunction ;

Complexity of switch statement m_2'< complexity of myfunction;

So the induction hypothesis can be applied.

The runtime of myfunction $A = O(\sum_{i=0}^{n}(abs(\arg_i) + f(m_1') + \sum_{i=0}^{n} abs(\arg_i)^{f(m_2')} + f(m_2'))$

Let's use g to represent $\sum_{i=0}^{n} abs(\arg_i)^{f(m_1')}$, the runtime of myfunction A =

$O(g^{2^{m_1'}} + 2^{m_1'} + g^{2^{m_2'}} + 2^{m_2'})$

Then we have $A \le O(g^{2^{m_1'}+2^{m_{12}'}} + 2^{m_1'+m_{12}'}) \le O(g^{2^{m_1'+m_2'}} + 2^{m_1'+m_2'})$

Let Let $m^{''} = m_1^{'} + m_2^{'}, f(m^{''}) = 2^{m_1^{'}+m_2^{'}}$, and the runtime of the function is bounded by $O(g^{2^{m_1^{'}+m_2^{'}}} + 2^{m_1^{'}+m_2^{'}})$. So induction hypothesis holds. This completes all cases for the theorem, so the theorem is true.

5.4 Simulating polynomial in the argument time Turing Machines

In this section we show any Turing Machine running in polynomial time in its inputs can be simulated by a function that our preprocessor will verify as polynomial. We make the following assumptions on our proof: (1) The Turing Machine never uses more tape squares than can be indexed by one int (roughly, 2^{32} squares) and the total number of steps of the computation can be stored in one int. (2) The output of the computation is the contents of the tape when the halt state is entered. (3) The Turing Machine only has one tape. (4) The alphabet of the Turing Machine only has two symbols 0,1 (and of course blank). The last two restrictions are not really a restriction as one can simulated a k-tape machine with a one tape one with at most a quadratic slowdown and similar simulate larger alphabets with smaller ones.

Theorem 2

Let P be a Turing Machine which on all inputs x runs in time bounded by x^k for some fixed k and otherwise restricted as above. Then there is a function f_P that our preprocessor will validate as polynomial such that on input and outputs satisfying the above restrictions f_P will output the same value as P.

5.4.1 Proof

Let P be a Turing Machine as above. Its input given the restrictions above will be a string of 0's and 1's. We assume these are passed to our polynomial function as an array of int's called input. The output will be returned by reference in an array called output which we assume can hold all of the tape squares needed for the computation. Below is a skeleton of what our function will look like:

```
polynomial  void  f_P(int[] input, int[] output)
{

        int len, i, maxtime, state;
        int head_pos;
        len = input.len;

        for ( i = len-1; i>=0; i--)
        {
                output[i] = input[i];
                maxtime = 2*maxtime + input[i];
        }
        /*new now use the variable output as the tape to do our simulation
        on */

        maxtime = maxtime * … *maxtime; //k th power
        head_pos =0;
        for( i = maxtime; i>0; i--)
        {
                switch (state)
                {
                        case 0:
                                if (output[head_pos] == 0)
                                {
                                state = new state;
                                output[head_pos] = new value;
                                head_pos = new_head_pos;
                                }

                                else if (output[head_pos] == 1)
                                {
                                state = new state;
                                output[head_pos] = new value;
                                head_pos = new_head_pos;
                                }
                                elseif (output[head_pos] == ' ')
                                {
                                state = new state;
                                output[head_pos] = new value;
                                head_pos = new_head_pos;
```

```
                                        }
                                        break;

                                case 1:
                                         if (output[head_pos] == 0)
                                        {
                                        state = new state;
                                        output[head_pos] = new value;
                                        head_pos = new_head_pos;
                                        }

                                        else if (output[head_pos] == 1)
                                        {
                                        state = new state;
                                        output[head_pos] = new value;
                                        head_pos = new_head_pos;
                                        }
                                        elseif (output[head_pos] == ' ')
                                        {
                                        state = new state;
                                        output[head_pos] = new value;
                                        head_pos = new_head_pos;
                                        }



                                //… one case for every state of P
                        }

                }
        }
```

In the above we assume, that we can represent states using ints (i.e., there are less than 2^{32} states in P). We do know the total number of states is finite so we could modify the above and use finitely many int's if there were more states than this. The purpose of the switch statement is to handle what P does when it is in a given state reading a given symbol off the tape.

Using the above function we can simulate P and so the theorem holds.

6. Conclusions

A Feasible C preprocessor was designed and implemented in this project. It parses our Feasible C polynomial function, and outputs normal C codes. If a Feasible C polynomial function is parsed without error, then this function is polynomial in its inputs.

In this project, given the assumption of assuming it takes OS effectively constant time to execute certain operations such as creation of local variable, we have shown that any Feasible C function parsed successfully by this preprocessor is polynomial on its input arguments. Within our Feasible C, we can also simulate any polynomial time Turing Machine. We have also shown that our preprocessor can verify that the standard implementations of sorting algorithms are p-time.

This preprocessor is implemented using Lex and Yacc. A hash table was used to as the symbols table. Dr. Louden has pointed out that the way our preprocessor handles *while* loop is better than the way it handles for loop. In the future we can improve our project by applying similar restrictions on *for* loops. It would be nice to extend our preprocessor so that it can also allow more data types such as recursive struct's. Hofmann [H02] gives a characterization of EXPTIME ($2^{p(n)}$, p(n) is a polynomial function of n) which allows computations over a variety of data structures such as lists or trees in a functional setting. So it should be possible to modify it in a C setting. However, this requires us to explore the possibility of having restrictions on recursive structs in our Feasible C so it won't affect our theorem.

7. References

[B88] Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language. 1988.

[B93] Bjarne Stroustrup. The C++ Programming Language. 1993.

[BC92] Stephen Bellantoni and Stephen Cook. New recursion-theoretic characterization of the polytime functions. Computational Complexity, 2:97-110, 1992.

[C99] P.Clote. Computation models and function algebras , Handbook of Computability Theory, ed. E.R. Griffor, Elsevier Science B.V. (1999). pp. 589--681.

[C64] Cobham, A. The intrinsic computational difficulty of functions. PROC. 1964 INTERNAT. CONGR. FOR LOGIC, METH., AND PHILO. OF SCI., North-Holland, Amsterdam, pp. 24--30.

[H02] Martin Hofmann. The strength of non-size increasing computation. Proceedings of the 29th ACM on Principles of programming languages. p. 260--269. ACM press. 2002.

[L93] Daniel Leivant. Stratified functional programs and computational complexity. In Twentieth Annual ACM on Principles of Programming Languages, pages 325-333, ACM Press.1993.

[LMB92] Lex & Yacc. Levine, Mason, and Brown. O'Reilly & Associates, Inc. 1992.

[Pap94] Computational Complexity. C.Papapdimitriou. Addison Wesley. 1994.