Amorphous Slicing of C Programs with TXL

Final Project Report

By

KANAGASABAI SRISKANTHAVERL

Under the
Supervision of

PROF. MARK HARMAN

SEPTEMBER 2007

MSC ADVANCED COMPUTING
DEPARTMENT OF COMPUTER SCIENCE
KING'S COLLEGE LONDON

TABLE OF CONTENTS

Abstract

Slicing can be defined as simplifying programs by eliminating irrelevant statements for a slicing criterion, and produces executable and semantically equivalent program with respect to the slicing criterion. Slicing is also a technique of visualizing dependencies and restricting attentions to just the components of the program relevant to the slicing criterion. Amorphous slicing is one of the many approaches for slicing, which simplifies a program while preserving a projection of its semantics. Other slicing techniques includes dynamic slicing, inter-procedural slicing, conditional slicing etc. Unlike a syntax-preserved slicing amorphous slicing can apply any simplifying transformation to produce slices, provided the semantic projection is preserved. Therefore it produces thinner and simpler slices. Push transformation, expression simplification, pointer elimination, de-modularizing complex statements are few techniques that an amorphous slicer can make use of.

Though slicing is a relatively young research area in the field of software engineering, considerable works have been done on building conceptual background and algorithms. However there is no previous work has been done on implementing amorphous slicing in TXL, a rule based language for program transformation. Programming with TXL often required twisting the algorithms designed for procedural languages. This project is an effort to implement an amorphous slicer in TXL which make use of amorphous slicing techniques like push transformation, expression simplification, pointer elimination and kill assignments.

The slicer implemented as a part of this project, is capable of applying amorphous slicing techniques such as push transformation, expression simplification, pointer elimination and kill assignments. The implantation anyway is not supporting each and every syntaxes of C language. As TXL uses pattern matching and replacing strategy for program transformation, it is required to write rules for every single possible syntax. The implementation supports as much as possible syntaxes to cover all scenarios discussed in this report. Irrespective of such limitations, the slicer is producing good results, with functional accuracy and slices downsized on average up to 35% of input program.

1 INTRODUCTION

Program slicing is a fundamental operation for many software engineering tools and also is a technique of visualizing dependencies and restricting attentions to just the projection of the program relevant to the computation of the slicing criterion. Slicing could be defined as the process of finding all the statements in a program that directly or indirectly affect the computation of the value of a variable at a particular line number. The thinner program constructed with the statements selected is called the *slice* of the program with respect to the slicing criterion. The process of slicing deletes parts of the program those can be determined to have no effect upon the semantics of interest.

Program slices, as originally introduced by Weiser, are now called executable backward static slices; *executable* because the slice is required to be an executable program; *backwards* because of the direction edges are traversed when the slice is computed using a dependence graph; and finally, *static* because they are computed as the solution to a static analysis problem i.e., without considering the programs input.

In the software industry, CASE (Computer Aided Software Engineering) tools, play a major role for efficient and accurate software development. In addition to the traditional software development tools like, Integrated Development Environment, Design recovery, walk-through debugging etc, program slicing also could be used in many ways in software engineering processes. The utility and power of program slicing comes from the ability to assist in tedious and error prone tasks such as program debugging, testing, parallelization, integration, software safety, understanding, software maintenance, and software metrics. Slicing does this by extracting an algorithm whose computation may be scattered through out a program from intervening irrelevant statements. Consequently, it should be easier for a programmer interested in a subset of the program's behavior to understand the slice.

1.1 Definitions

Definition: A *slicing Criterion* for a program slice is a tuple *<i, V>* where *i* is a statement in the program and V is a sub-set of the program variables. A slice of a program is computed with respect to v, at statement i.[1]

Definition: A *program slice* of a program P for slicing criterion, <i, V> is an executable program obtained by deleting zero or more statements from P such that the values of the variables in V are the same when execution reaches statement I of both P and the slice of P. [1]

Definition: q is an *amorphous slice* of program p with respect to a slicing criterion <i, V> if and only if it is the simplest program of p while it is equivalent to the program p with respect to the slicing criterion considered. [2]

Definition: *Refs(n)* is the set of variables referenced (the variable's value is used) at statement n. [1]

Definition: *Succ(n*) is the set of statements that could be executed immediately after statement n [1]

Definition: *kill(n)* is an assignment statement that overrides the value of a variable previously assigned at statement n.

Definition: *EffectiveKill(n)* is a kill(n) at statement m which make the value assignment at statement n redundant, as it is not being referenced at any statements in Succ(n) – Succ(m)

Definition: *Redundant assignment* is an assignment statement at line n, where there is an *EffectiveKill(n)* at a location m in Succ(n).

Definition: *assignment variable* in an assignment statement is the variable whose value is modified to a new expression.

Definition: *assignment expression* in an assignment statement is the expression in the right hand side of the assignment operator, which is assigned to the *assignment variable*.

1.2 Background

Slicing has two approaches depends on the purpose of slicing and how to do it; it is either backward slicing or forward slicing. Backward slicing projects all the program statements affect a given point in the program. On the other hand, a forward slicing projects all the program statements that are affected by a given point. A backward slicing is more powerful and wide application scenarios than a forward slicing, and a forward slicing is the one generates a complete slice of the program that is able to compile and execute. This project concentrates on backward slicing, and more accurately on amorphous slicing.

```
int main() {
     int sum = 0;
     int i = 1;
     while ( i < 11) {
          sum = sum + i;
          i = i + 1;
     }

     printf (“%d\n”, sum);
     printf (“%d\n”, i);
}
```

Backward slice from statement `printf(“%d\n”, i)`

```
int main() {
     int sum = 0;
     int i = 1;
     while ( i < 11) {
          sum = sum + i;
          i = i + 1;
     }

     printf (“%d\n”, sum);
     printf (“%d\n”, i);
}
```

Forward slice from statement `sum=0`

Figure 1-1

In Figure 1.1, the program consist of two ‘computational threads’ one computes the value of variable occurrence `‘sum’`, and another one computes the value for `‘n’`. A backward slicing respect to the statement `printf(“%d\n”, i);` shown in the left hand side, generates a slice that affects the computation of variable occurrence `‘n’`(highlighted in bold italic). On the right hand side, a forward slicing respect to the statement `‘sum = 0’` generates a slice consist of statement that are affected by the statement in concern.

There are many forms of slices such as static slice, dynamic slice, conditioned slice, amorphous slice etc.

A *static slice* is constructed by deleting irrelevant parts of the programs to reach a projection of it, which includes all the statements relevant to preserve the semantic meaning of the program in view of slicing criterion. A static slicer of a program p is computed based on the information statically available on the program; since the program is analyzed statically, the slicing operation depends on the syntactical analysis of the program.

On the other hand, a *dynamic slice* is making use of the input values to the program to generate a much better and thinner slice than a static slice. A dynamic slice preserved the semantic meaning of the original program with respect to the slicing criterion and to the input values considered, whereas a static slicer preserved the semantics of the program for all possible input values. A dynamic slice is much more convenient for debugging for identified malfunctioning of a program for certain input values.

A *conditional slicing* is again slicing a program for a subset of the universal set of inputs. However unlike a dynamic slice, which generate a slice for one particular instance of input parameters, a conditional slice, do slicing based on predefined conditions to the input variables. For example a conditional slicing program can be input with a condition x = 2 * y, to generate a slice for all inputs of x and y satisfy the condition x = 2 * y.

All of the above slicing approaches are 'syntax preserved slicing', preserving the syntax of the original program in the slice as it generates the slices by simply deleting irrelevant statements and left the relevant statements untouched. Amorphous slicing on the other hand is constructed by simplify the program with a slicing criterion by any kind of program transformations, while it ensure the semantic of the original program with respect to the slicing criterion is preserved in the slice.

1.3 Applications of Slicing

Slicing is a broadly applicable static program-analysis technique. Applications of slicing include program understanding, debugging, testing, parallelization, re-engineering, maintenance, etc. Slicing can be used to understand the code by producing slices on various slicing criterion and review them. Slicing is useful to isolate "computational threads", which helps in identifying logical components; threads can be extracted and either replaced or used to create new programs; called program restructuring. Further more slices are generated as specialized programs and which could be reused later on another application; in this way slicing is used for generating reusable codes. Instead of including a complete package for reusing in another program, slicing allows to identify only those parts to be included to the program. Another interesting and critical application of slicing is safety/security validation; slicing supports software inspection for validation of safety critical and security-critical software. It validates for any improper information flow from a high-security input to a low-security output. Considering these innumerable advantages, any efforts to improve and push forward the current slicing technique is worthy and has practical values.

1.4 Amorphous Slicing

The traditional slicing technique is based on the syntactical meaning of the program and a statement in the original program is either appears in the slice as it is or not appears at all; which is called as syntax-preserved slicing; in other words the slice is a projection of the original program. On the other hand amorphous slicing is a concept of slicing a program without being bound to any syntactical restrictions. Amorphous slicing may use any simplifying transformation which preserves these semantic projects, thereby improving upon the simplification power of traditional slicing. Amorphous slicing almost always generates a thinner slice than syntax-preserved slicing. Amorphous slicing is more suitable for program comprehension problem while syntax-preserved slicing is more suited for debugging.

```
for(i = 0, sum = a[0],
           biggest = sum;
           i < 19;
           sum += a[++i]){
      if(a[i + 1]  > biggest)
           biggest = a [i +
1]
}
average = sum/20;
printf (“%d\n”, biggest);
```

Original program

```
for(i = 1, biggest = a[0];
                i < 20; ++i)
     if( a[i]  > biggest )
          biggest = a [i + 1]

printf (“%d\n”, biggest);
```

Amorphous slicing for statement `print (“%d\n”, biggest);`

Figure 1-2

In Figure 1-2, the right hand side is the amorphous slice generated on the program in the left hand side, respect to the statement `‘printf(“%d\n”, biggest)’.` In this example two computational threads are found; one for `‘sum’` and another for `‘biggest’`. In the first phase a syntax-preserved approach is considered and any statements irrelevant to the computation of the slicing variable are removed. In the second phase an amorphous slicing approach is applied; few statements get reduced to single statement, loop invariant get adjusted without alter the semantic of the program. The construction of amorphous slicing is considered harder than syntax-preserved slicing, because any transformation can potentially be applied in amorphous slicing.

2 LITERATURE REVIEW

2.1 TXL & C Grammar structure

TXL is the language of choice to implement amorphous slicing for this project, as it is a unique programming language specifically designed to support source code analysis and source transformation tasks. It is a hybrid of functional and rule based language. TXL is unique in that it has a pure functional super structure that provides scoping, abstraction, recursion, pattern search, implied iteration etc. which are heavily demanded for program analysis and transformation processes. The TXL paradigm consists of parsing the input text into a tree structure, transforming the tree to create a new structure tree, and unparsing the new tree to a new output text. TXL best suites for the tasks of programming language processing such as rapid prototyping of a new language parsers, or language extensions, source to source program translation, Local and global source code optimization etc, program analyzing and instrumentation tasks like program structural comparison, normalization, program restructuring, remodularization, program understanding and visualization code slicing etc., software engineering tasks like design recovery, architecture extraction from source, security analysis etc.[10]

TXL Transformation process

TXL operates in three phases such as parsing phase, transformation phase, and unparsing phase. In the parsing phase, the parser takes the entire input, tokenize it, and parse it according to the TXL's program's grammar definition. This will produce a parse tree. Once the input is converted into a parse tree, the transformation rules written for a specific intended purpose by the TXL programmer, will be applied taking the parsed tree as the input. During this phase the input parse tree is transformed in to another tree according to what is specified in the transformation rules. Once the transformation is done, the transformed tree structure is unparsed to generate the output program.

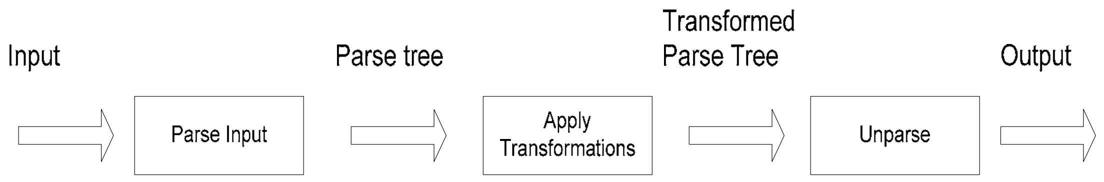


Figure 2-1

C Grammar Structure

TXL build a tree structure internally based on the grammar of the language defined with TXL, while parsing the input program. As an example a simplified version of C grammar defined by TXL is explained as follows.

On the very low level, each token parsed in are identified as a [string], [id], [number], [assignment_operator], [binary_operator], [unary_operator] or [key]. Any string constants in the program is parsed to a [string] literal, variables, function names etc are identified as [id] literals, numerical values in the expressions are identified as [number] literals, operators in the set {`=, *=, /=, %= , += , -= , >>=, <<=, &=, ^=, |=`} are identified as [assignment_operator]s, operators in the set { `+, -, *, /, %, ==, !=, <, >, <=,`

`>=, ||, &&, |, ^, &, <<, >>, <, >}` are identified as [binary_operator]s, operators in the set `{ *, &, +, -, !, ~, ++, --}` are identified as [unary_operator]s and all the key words in C are identified as [key]s.

The literals [reference], [unary_expression], [binary_expression], [assignment_expression], [expression], [declaration], [statement], [function_definition] are further defined based on the basic literals discussed above.

A [reference] is either a [number], [string], or [id]; typically variable names, function names, numerical constants are identified as [reference]s by TXL C grammar. A [unary_expression] is defined as a [reference] or a [unary_operator] followed by a [unary_expression]. A single identifier, or a parenthesized expression, a numerical constant, or any of the above preceded with a unary operator are identified as [unary_expression]s. A [binary_expression] is either a single unary expression or a binary expression followed by a binary operator and a unary expression.
The following parse tree explains a part of the grammar structure got by parsing a binary expression ‘`2 * i + j`’.

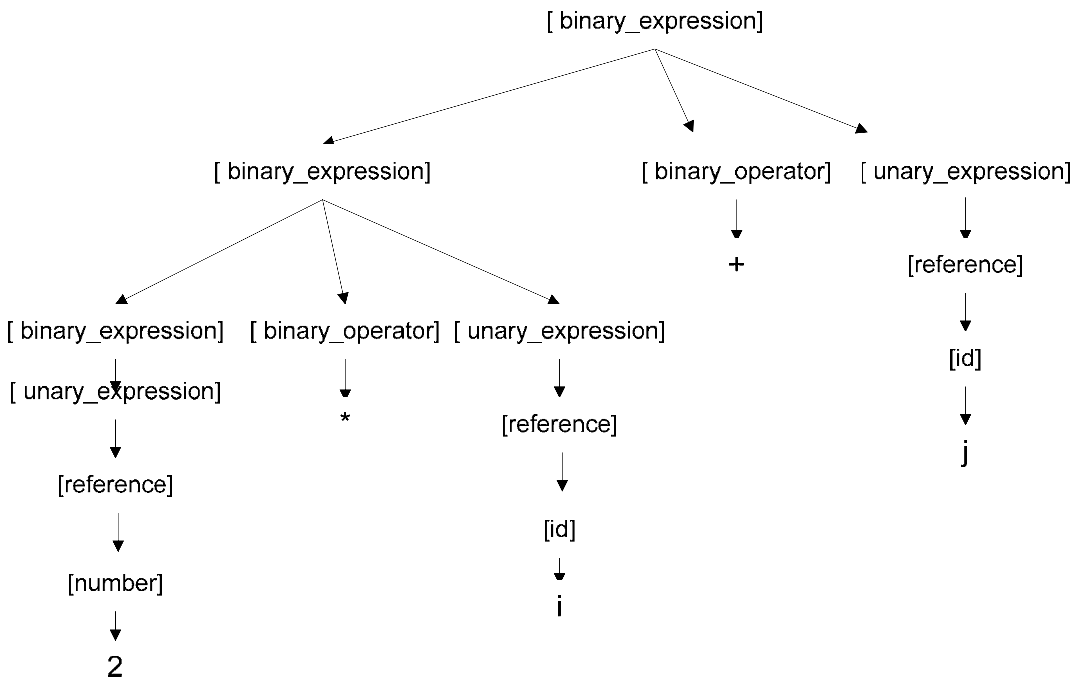


Figure 2-2

At the top level of C Grammar as defined in TXL, it defines [program], [declaration], [function_definition], [statement] etc. A [program] is defined as a sequence of [declaration]s and [function_definition]s. Variable declarations, function declarations, preprocessor declarations are identified as declarations and function definitions are identified as [function_definition]s. A [function-definition] is defined as a function header followed by a [compound_statement], where a [compound_statement] is a sequence of [statement]s and [declaration]s within a pair of curly braces {}.

For example a simple C program listed in figure 2.3 parsed into a TXL tree structure as shown in figure 2.4.

```
int i, j, k;
char *a, *b;
void main(){
      int p, q;
      p = 0;
      q = 2 * p;
      printf("%d, %d", p, q);
}
```

Figure 2-3

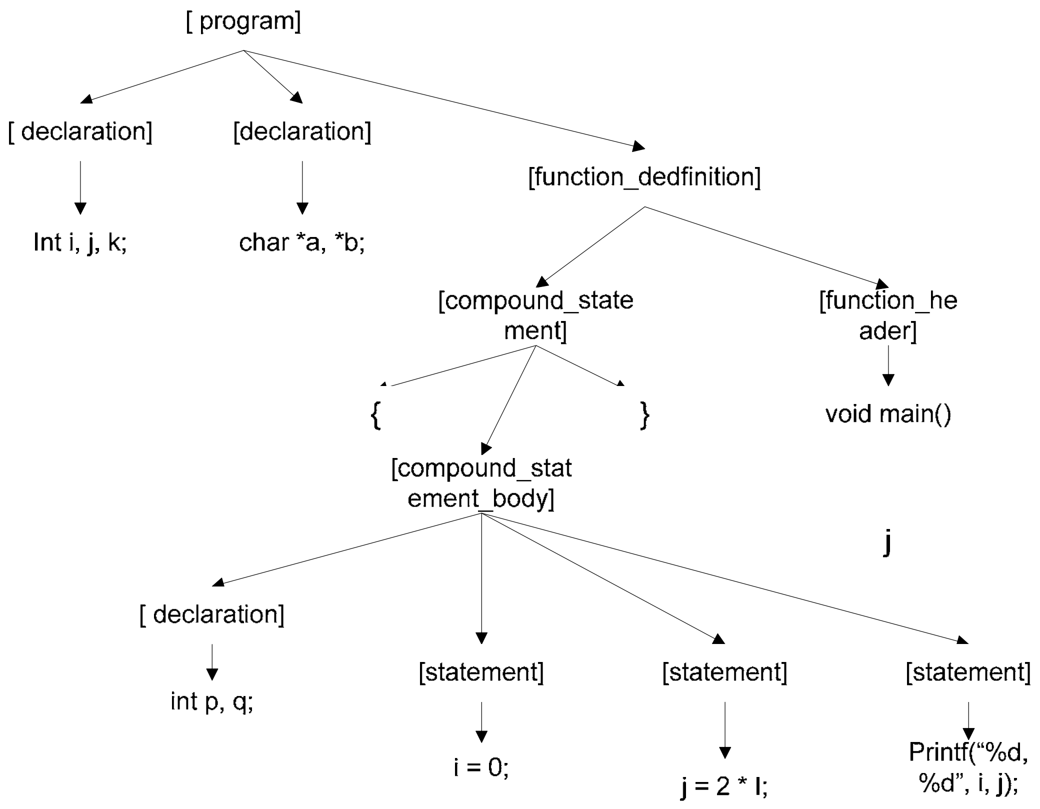


Figure 2-4

A [statement] is further defined as either a [expression_statement], [if_statement], [while_statement], [do_statement], [for_statement] etc, where each of them defines the specific syntax of their own. For example, a [while_statement] is defined as follows; which reflect the syntax of a while loop in C language.

```
define while_statement
    'while '( [expression] ') [statement]
end define
```

The complete definitions to C Grammar as defined in TXL and extensions and overrides to it as a part of this project can be found at the end of the report in Appendix A.

2.2 Dependency Analysis

A program is a collection of statements, the ordering and scheduling of which depends on dependency constraints. Dependencies are broadly classified into two categories.

Data dependencies: Also known as true dependency, occurs when an statement depends on the results of a previously executed statement. For example, consider the simple program listed in Figure 2.5.

```
1. void main(){
2.    int a, b;
3.    a = 5;
4.    b = a;
5.    c =  2 * b;
6. }
```

Figure 2-5

In this program, line 5 is truly dependent on line 4, as the final value of c depends on the instruction updating b. line 4 is truly dependent on line 3, as the final value of b depends on the instruction updating a. Since line 5 is truly dependent upon line 4 and line 4 is truly dependent on line 3, line 5 is also truly dependent on line 3.

Control Dependency: These are the dependencies which arise from the ordered flow of control in a program. The flow of control of a program is the order of the statements that appear in the program for most of the programming languages. However it could be disturbed by conditional statements, looping statements, jump statements etc.
Consider the program in Figure 2.6.

```
1. void main(){
2.    int i;
3.    scanf(“%d”, i);
4.    if ( i < 0)
5.          i = (-1) * i;
6.    else
7.          i =  2 * i;
8.    printf (“%d”, i);
9. }
```

Figure 2-6

Statement 5 and 7 has a control dependency on statement 4, as which statement to execute depends on the evaluation of conditional expression in statement 4.

Dependency Graph
Program dependencies are effectively represented using Dependency graphs, where Nodes as statements are directed arcs as the dependencies.

```
1. void main(){
2.    scanf("%d", t1);
3.    t2 = t1 + 4;
4.    t3 = t1 * 8;
5.    t4 = t1 – 4;
6.    t5 = t1 / 2;
7.    t6 = t2 * t3;
8.    t7 = t4 – t5;
9.    t8 = t6 * t7;
10. }
```

Figure 2-7

Data dependency graph for the program in Figure 2-7

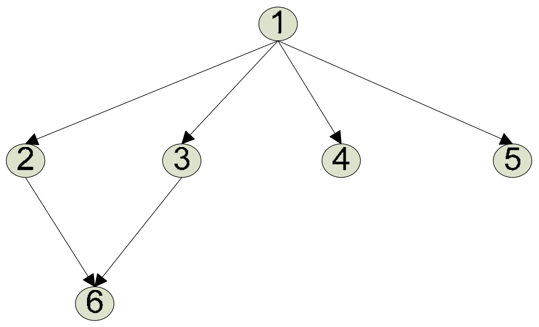


Figure 2-8

Control dependency graph for the program in Fig 2-7

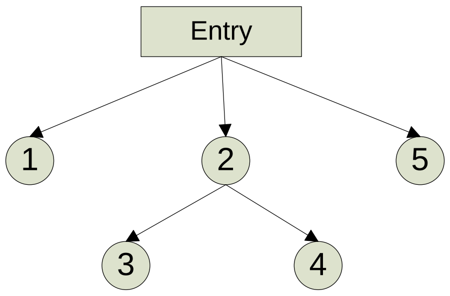


Figure 2-9

The data dependencies and control dependencies are together generate the complete program dependency graph.

2.3 Pointer Analysis

A slicing tool may not be practically much useful unless otherwise it handles the real world programming features such as pointers. A programming language such as C, where the power of C resides with this feature, it is virtually impossible to find a real world program which doesn't use pointers. When it become slicing in the presence of pointers, there are few problems or ambiguities arise with deciding what is really meant to be sliced. For example,

slicing for a pointer variable, at statement n may mean to include all the statements that are relevant to the computation of the value at the memory location pointed by the slicing pointer variable/s. Or it may mean to include all the statements relevant to the computation of a class of variables that can be referenced by the slicing pointer variable/s. The slicing criterion should be precise enough to eliminate this kind of ambiguities.
There can be two kinds of addresses in a program; addresses to static objects and addresses to dynamic objects. The essential difference is that a program with static object pointers can be eliminated while preserving the semantic of the program; however it is not always possible to eliminate pointers to dynamic objects, as pointers can be the only way to access to those objects, in some programming languages such as C.

Secondly, there are instances, where a slice of a program involving pointers, may become non-executable. This could be better explained by the following example.

```
1 #include <stdio.h>
2 void main() {
3     int a, b, s, u, x, m;
4     int *w, *y, **z;
5
6     s = 1;
7     a = 2;
8     b = 3;
9     scanf("%d %d", &x, &u);
10
11    if(x) y = &a;
12    else y = &b;
13
14    if(u) z = &y;
15    else z = &w;
16
17    w = &s;
18    **z = 4;
19
20    printf("s %d \n", s);
21}
```

Figure 2-10

In the sample program in Figure 2-10, consider a slice on s at line 20, which should include statements {1, 2, 6, 9, 14, 15, 17, 18, 21}. However by leaving lines 11 and 12 out of the slice, there is a risk of producing a non-executable slice. That is if u is true, then z is pointing to y where y doesn't point to a valid memory location, hence produce a run-time error at line 18. This problem can be overcome by either include all the other statements such as 11 and 12 which are relevant to produce an executable slice, or by guard the vulnerable statements with appropriate conditions.

There are two approaches to do slicing on a program with pointers. First analyzing the data dependency with the pointer variables and generate the slice by including all the objects as well as pointers that have dependency with the slicing criterion. A second approach could be eliminating the pointers from the program in the first phase and slice on the resulting program which doesn’t involve with any pointers.

Analyzing data dependency of the program in Fig 2-10 for the slicing criterion, it can be concluded that points-to(w) = {s}, points-to(z) = {y, w}, points-to(y) = {a, b}. Here points-to(x) = {a, b} means that x is potentially points to a and b.

Slicing considering control and data dependencies, the dependency analysis can be as follows. `**z` at line 18 potentially refer `s` (`z` potentially refer to `w`, and `w` refers to `s`); hence line 18 will become part of the slice. Now line 18 uses `**z` a double pointer dereferencing, the memory locations defined by `*z` and `z`, those locations also need to be included in the slice. Since `*z` potentially refer to memory locations {y, z}, and `z` also being referenced in line 18, {z, y, w} are potentially used by the slice, and the memory locations of those need to be included. The variables `z`, `y` and `w` are defined at lines 14, 15, 11, 12 and 17; hence these lines need to be included in the slice. Further lines 11 and 14 have dependency with line 9. Finally, since line 18 is only a potential kill assignment to `s`, the initial assignment to s at line 6 also have to be part of the slice, as `s` is used in line 20 of the slice. That is, lines {1, 2, 3, 4, 6, 9, 11, 12, 14, 15, 17, 18, 20, 21} are the possible slice of the program in Fig 2-10 for the slicing criterion <20, {s}>.

The second approach is eliminating the pointers first, by replacing all pointer dereferences with the target variables, guarded with appropriate conditions. Analyzing the program in Figure 2-10 in this approach goes as follows. First the definition of `z` at line 14 and 15 is replaced the pointer dereferencing at line 18; line 18 is replaced with the following lines.

```
if(u) *y = 4;
else *w = 4;
```

Further, the definition to `y` at line 11 and 12 are replaced on the above lines to produce the following lines. Also the definition of w at line 17 is replaced for its dereferences.

```
if(u)
      if(x) a = 4;
      else b = 4;
else
      s = 4;
```

Once all the pointer dereferences are eliminated, the pointer definitions can be eliminated. Now the slice will include {1, 2, 3, 6, 9, 20, 21} from the original program and ‘`if(!u) s = 4;`’ at line 18. Which is much simpler and thinner compared to the first approach

3 SLICING ALGORITHMS

3.1 Static Slicing Algorithm

It is advantageous to eliminate any irrelevant statements in the program which could be identified by the syntax-preserved static slicer as before applying an amorphous slicing algorithm. The static slicer module developed in this project, implements rules to identify statements directly or indirectly affecting the slicing criterion. It is capable of tracing dependencies in a single block of statements, conditional statements, and also looping statements. Though the implementation is concentrated on while loops, the same algorithm can be applied to other looping statements such as for loop, do-while loop etc. in C language.

When it becomes programming with TXL, which doesn’t have a rich data structures (of course it is not what TXL is for), a different approach to represent the dependency had to be explored. Further for the purpose of slicing what it required is the ultimate dependency of the statements in the program with the slicing criterion. It is not our concern the exact dependency path between slicing criterion and the dependent statements. Taking advantage of the TXL's internal tree representation of the program, the algorithm discussed here, keep the dependency information in the tree structure itself by marking those statements as required.

The standard C grammar available with TXL, defines expression_statement, for_statement, while_statement, do_statement etc. as valid forms of statements. In addition to them, marked_statement is defined, which is also a form of statement. A marked_statement defined as a valid statement enclosed within an xml tag and closing tag.

```
define marked_statement
      [xml_tag] [statement] [xml_end]
end define

define xml_tag
      < [SPOFF] [id] > [SPON]
end define

define xml_end
      < [SPOFF] / [id] > [SPON]
end define
```

In the set of statements shown in Figure 3-1, dependencies on variable `t6` can be represented in the program itself, by converting the dependent statements into marked_statements.

```
1. <dep>scanf(“%d”, t1); </dep>
2. <dep>t2 = t1 + 4; </dep>
3. <dep> t3 = t1 * 8; </dep>
4. t4 = t1 – 4;
5. t5 = t1 / 2;
6. <dep>t6 = t2 * t3;</dep>
7. t7 = t4 – t5;
8. t8 = t6 * t7;
```

Figure 3-1

Since `t6` depends on `t2` and `t3`, `t2` depends on `t1`, `t3` depends on `t1`, statements that assign values to `t6`, `t2`, `t3`, `t1` are marked with <dep> </dep> to trace the dependency details.

The syntax preserved slicing algorithm can be sub divided into following sub algorithms.

1. Back propagate data dependencies from the slicing point up to the beginning of the program
2. Propagate dependencies through looping statements
3. Remove statements those neither have dependency with the slicing variables, nor a block statement have a sub-statement having dependency with the slicing variable
4. Eliminate any empty block statements.

In the first phase of the algorithm, it propagates the dependency backward from the slicing point to the top of the program; all the statements having dependency are converted into marked_statements. The algorithm examines the statements in the program from top to bottom; for each assignment statements encountered, looking for any marked_statement down the program having dependency with this assigning variable; if it is so, then the algorithm converts the currently examining statement also into a marked_statement. This process of dependency analysis is being done again and again from top to bottom, until there is no more statement having dependency with slicing criterion, left unmarked. In the very first, only the slicing statement is marked (by the user to indicate the slicing point). By the end of the first iteration, all the assignment statements that have direct dependency with the slicing statement will be turned into a marked_statement. In other words, the dependency graph is constructed for the dependencies of path length equals to one. In the same manner the following iterations each expands the dependency graph by one additional level, until the complete dependency graph is constructed in the form of marked_statements. The outline of the algorithm is listed in Figure 3-2.

```
PROPAGATE_DEPENDENCY ( PROGRAM)
     DO
          for each statement S in PROGRAM
               M is the statements follows S
               If (S is not marked)
                    deconstruct S
                    'X = Exp'
                    if ( USED_IN_MARKED_STMT(X, M))
                         mark S
     UNTIL (no additional statement(s) marked)
END

USED_IN_MARKED_STMT(X, BLOCK)
     for each statement S in BLOCK
          if (S is marked)
               deconstruct S; 'Y = Exp'
               if ( X found in Exp)
                    return true;
     return false;
END
```

Figure 3-2

Consider the following sample program.

```
1     #include <stdio.h>
2     void main(){
3           int i, j, k, sum, avg;
4           count = 0;
5           <mark>i = 5; </mark>
6           <mark>j = 10; </mark>
7           <mark>sum += i; </mark>
8           count = count + 1;

9           <mark>sum += j; </mark>
10          count = count + 1;
11          k = 2 * i;
12          avg = sum / count;
13          <mark>printf("sum = %d", sum);</mark>
14          printf("avg = %d", avg);
15    }
```

The slicing point is marked first at line 13, which prints the value of variable `sum` at the end of the program. In the first iteration the algorithm marks statements 9 & 7, as they have direct data dependency with `sum`. The second iteration identifies line 5 has dependency with line 7, line 6 has dependency with line 9. The third iteration doesn't found any new statements having dependency with any of the marked statements; the algorithm terminates at this point. The variable declarations in line 3 is not considered as a statement in the C grammar, it is neither marked nor eliminated.

The second phase of the algorithm, handles the effect of control dependencies by the existence of any looping statements, such as while loop. This works on the following two facts; first, any statement inside a loop may get executed after any other statement inside the loop; second, any statement that modify the looping conditional expression have dependency with all other statements inside the loop. The algorithm process one looping statement at a time, marking all the statements inside the loop, those have dependency with any others inside. Finally if any statement inside the loop having dependency with the slicing variable (already marked), then mark all the statements inside those modify the looping conditional expression.

Consider the while loop in Figure 3-3, which is a part of a program computing the Fibonacci number of first N numbers and also prints the maximum prime less than N. Assume line 4 is marked already in the first phase by back propagating algorithm. Since line 5 has dependency with line 4; line 3 has dependency with line 5, these additional lines are marked for dependency. Since there are statements inside the loop, having dependency with the slicing variables, line 8, which modify the looping conditional expression, also marked for dependency.

```
while( i <= N){
        printf("\t%d \t   %d\n", i, current);

        twoaway = current+next;

        <mark>current = next; <mark>
        next = twoaway;

        is_prime = 1; //true
        curr_prime = i;

        i = i +1;
}
```

Figure 3-3

In the third phase of the algorithm, all the unmarked statements in the program are eliminated and left with the statements relevant to the slice; the block statements such as while statement, if statement etc, are additionally checked for not included any marked statements inside before being eliminated. Variable declarations, function headers, preprocessor statements are not statements under the TXL grammar for C. Therefore those lines will remain exist in the slice.

Finally all the marked statements are turned back into unmarked, and any unused variable declarations are eliminated to generate the final version of the static slice for the given slicing criterion.

3.2 Push Transformation

Push transformation is an amorphous slicing technique to down size the line count of the slice, by way of pushing assignment expression to variables wherever possible, to its references down the program.

For an example consider this simple C program.

```
1    void main(){
2            int i, j;
3            i = 3;
4            j = 2 * i + 4;
5            printf(“%d”, j);
6    }
```

Figure 3-4

In this example, in line 3 `i` is assigned to a value of `3`; and line 4 assigns an expression to `j` which depends on `i`. A push transformation replace the instances of i down the program with the expression `i` is assigned to, which is `’3’`. Therefore the push transformation replace line 4 with `j = 2*3 + 4`. Further the assignment to `j` in line 4 is pushed down in statement 5 to get `printf(“%d”, 2*3 + 4)`. By applying the expression simplifying algorithm discussed in section 3.3 in this report, `2* 3 + 4` is simplified to `10`, and the expected output will be as shown below in step by step.

```
void main(){
     int i, j;
     i = 3;
     j = 2 * 3 + 4;
     Printf(“%d”,
j);
}
```

```
void main(){
     int i, j;
     i = 3;
     j = 2 * 3 + 4;
     printf(“%d”,
2*3 + 4);
}
```

```
void main(){
     printf(“%d”,
10);
}
```

Figure 3-5

However, a reassignment to either the pushing variable or any variable in the replacement expression will result the push transformation further down the program incorrect. The example shown in Figure 3-6 explains this scenario better.

```
1 void main(){
2    int i, j, k;
3    scanf(“%d”, k);
4    i = 3;
5    j = 2 * k + 4;
6    printf(“%d”, i);

7    i = 7;
8    k = 3 * i;
9    Printf(“%d”, j);
10   printf(“%d”, k);
11 }
```

Figure 3-6

The assignment statement at line 4, `i = 3`, can be pushed down until the value of i has been reassigned. In the example line 4 can be pushed down up to line 7, replacing statement 6 with `printf(“%d”, 3);` . Further, assignment statement at line 7 can be pushed down until the end of the program, as there is no reassignment to i after line 6. In case of statement `j = 2 * k + 4;` in line 5, the expression `2 * k + 4`, can be pushed down the program until a reassignment to either `j` or any variables in the replacement expression `2 * k + 4`, which is {k}. Since there is a reassignment to `k` in line 8, line 5 can be pushed down up to line 8. It is incorrect to push the expression `2*k+4` to the instance of `j` at `printf(“%d”, j);` in line 9; because, the value of `j` at line 5 is computed with the value of k input by the user at line 3, this is the value of `j` that is expected to appear in line 9. On the other hand, if the expression `2*k+4` at line 5 is pushed to line 9 to get `printf(“%d”, 2*k + 4);`, the value of the expression in the `printf` statement is evaluated based on the value of `k` that has been reassigned to `k=3*i;` at line 8, which is incorrect. The expected push transformed slice of the program listed in figure 3-6 is as shown in Figure 3-7.

```
1 void main(){
2    int i, j, k;
3    scanf(“%d”, k);

5    j = 2 * k + 4;
6    printf(“%d”, 3);


8    k = 3 * 7;
9    Printf(“%d”, j);
10   printf(“%d”, 3 * 7);
11 }
```

Figure 3-7

Push transformation in the presence of conditional statements

As discussed before, an assignment to a variable can be pushed down the program until another statement reassigns the variable to a different value. This statement holds true even for a conditional reassignment. However pushing down the reassigned value further down is restricted within the scope of the conditional statement itself. Any reference to this variable further down the conditional reassignment can't be replaced with the conditionally assigned value, as it can't be determined at compile time whether the conditional reassignment will be executed.

```
1 void main(){
2    int i, j, k, N;
3    scanf(“%d”, N);
4    i = N;
5    j = 2;
6    printf("%d, %d", i, j);
7    scanf(“%d”, k);
8    if( k > 0){
9         i = 2 * N;
10        printf("%d, %d", i, j);
11   }
12   printf("%d, %d", i, j);
13 }
```

Figure 3-8

Consider the sample program in Figure 3-8. At line 4, `i` is assigned to `N`, which can be pushed down until it get reassigned in line 9. Hence line 6 will get replaced with `printf("%d, %d", N, j);`. Assignment to `j` in line 5 can be pushed down until the end of the program, as there is no reassignment to `j` down the program. The conditional assignment to `i` in line 9, can be pushed down only within the scope of the if statement. Hence line 10 will get replaced with `printf("%d, %d", 2*N, j);`. However at line 12, it is only at run time, the value of i can be determined, as the conditional expression at 8 may be true, which leads to the reassignment i `= 2 * N;`, otherwise the previous assignment at line 4, `i = N;` remains valid. That is the value of i at line 12 can't be determined by statically analyzing the program. The expected push transformed output of the above program may be similar to the one shown in Figure 3-9.

```
1 void main(){
2    int i, j, k, N;
3    scanf(“%d”, N);
4    i = N;
5    j = 2;
6    printf("%d, %d", N, 2);
7    scanf(“%d”, k);
8    if( k > 0){
9         i = 2 * N;
10        printf("%d, %d", 2*N, 2);
11   }
12   printf("%d, %d", i, 2);
13 }
```

Figure 3-9

Push Transformation with loop statements.

When it happens to any reassignment to a variable inside a loop, similar to conditional reassignment, the previous assignment can't be pushed further down the loop statement. In addition, it is incorrect to push down the previous assignment inside the loop, or the looping invariant, considering the fact that a reassignment inside a loop, affect the references to it in expressions anywhere in the loop and the looping invariant.

```
1  void main() {
2    int j, k, N;
3    scanf(“%d”, N);
4    j = N;
5    while(j > 0){
6         printf("Before: %d", j);
7         j = j – 2;
8         printf("After: %d", j);
9    }
10   printf(“Final: %d”, j);
11 }
```

Figure 3-10

Consider the sample program in Figure 3-10. The assignment to `j` in line 4, is reassigned at line 7, inside the while loop. Since statements 5, 6, 7, 8 (the whole body and the looping conditional expression of the while loop) may get executed after reassignment to j at line 7, obviously, assignment at line 4 can be pushed only up to line 4. Further, an assignment to a variable inside a while loop, can be pushed down the statements within the scope of the loop, as it is guaranteed that they are executed in order inside the loop.

However as a special case, if an assignment expression having reference to the assigning variable itself in an assignment statement (the case in line 7), those statements are not considered for push transformation because of its complicated behavior. To see the problem of pushing an assignment expressions of this kind consider the program in Figure 3-10 again. The assignment expression in `j = j – 2;` can be pushed into statement in line 8, provided assignment statement at line 7 is eliminated after being pushed. On the other hand, If line 7 is not eliminated the segment of the program appear as follows, will print the value at line 8 as j – 4 as it is reduced by 2 in line 7 and line 8 also prints (j -2), which is incorrect.

```
7    j = j – 2;
8    printf("After: %d", j -2);
```

It is also not possible to eliminate the assignment statement at line 7, after being pushed. Because the value of `j` is being used at lines 5, 6 and 10, which have dependency with line 7.

Algorithm

Programming with TXL, a rule based language; it required a different approach than other procedural languages. TXL allows applying any kind of transformations on a sub tree of a program, and it is tricky to apply a rule to a section of a tree, omitting some branches of it.

Considering this, the following mark and apply strategy is applied to implement push transformation in TXL.

```
PUSH(Stmts Statements)
begin
     deconstruct Stmts into
          S, first statement in Stmts
          More, rest of the statements in Stmts.

     if (First is NOT an assignment statement)
          return;

     DepIDs = ExtractVariables(S)

     MarkReAssignments(More, DepIDs)

     MarkStatementsFollowingReassign(More)

     PushOnUnmarked(S, More)

     UnMarkAll()

end
```

Since TXL is a rule based language, and recursively match all the patterns, the rule PUSH is applied to all the matching sequence of statements `Stmts` in the program. It should be noted that a set of statements {`S1, S2, S3, S4`} will have 4 patterns matching as set of statements; such as `{S1, S2, S3, S4}, {S2, S3, S4}, {S3, S4}` and `{S4}`. First the set of statements `Stmts` is deconstructed into `S`, the first Statement in the matched set `Stmts`, and `More` the rest of the statements in the set. If the first statement `S` is not an assignment statement, the rule returns. A successful set which has an assignment expression as the first statement is considered for a push transformation.

`ExtractVariables(S):` is a function, that extracts all the variables in the assignment expression, and return a list contains those variables and also the assigning variable of the assignment statement. For an assignment statement `p=3*k+i+j+3`, ExtractVariables() will return a list contains {p, k, i, j}.

`MarkReassignment(Stmts, DepIDs)`: Since an assignment to any of the variable in the list `DepIDs`, can stop the push operation on the statements following it, this function mark all the statements in `Stmts`, those assign values to the variables in the `DepIDs` list.

`MarkStatementsFollowingReassign(More):` An assignment can't be pushed on those statements following a reassignment. Therefore those statements are marked differently to exclude them from being replaced while applying the push transformation.

For a set of statements shown in Figure 3-11, with an assignment statement as the first statement, these functions mark them as shown in Fig 3-11 itself.

```
j = 6* i;
printf("%d", j);
if( true){
        printf("%d", j);
        <reassign>i = 3; </reassign>
        <mark>printf("%d", j);</mark>
}

<reassign>j = 5;</reassign>
<mark>printf("%d", i);</mark>

<mark>printf("%d", j);</mark>
```

Figure 3-11

Then `PushOnUnmarked()` replace all the instances of the target variable in the unmarked statements with the assignment expression.
Finally all the marked statements are unmarked before apply the push operation for another assignment statement.

Eliminating pushed assignments

If an assignment expression is pushed down to all the statements up to the reassign statement , and the reassign statement is in the same scope of the program, it is safe to eliminate the assignment expression after being pushed down. However, if the reassignment is a conditional reassignment or inside a loop statement, it is not guaranteed that the reassignment will be executed. Further in case of a reassignment inside a loop statement, the references to the assigning variable may not get replaced in all places. Therefore it is not possible to eliminate the assignment statements all the time after being pushed down. This algorithm does not eliminate any of the assignment expression from the program. However those statements that can be eliminated, will be identified as a redundant assignment, and eliminated. This is discussed in section 3.4.

Note: There are two scenarios to be carefully handled while marking for statements inappropriate for push operation. Consider the push transformation is applied on a sequence of statements S, and the assignment statement is A, the first statement in S. The First scenario is that, if the reassignment is not one of the statement in S, but a sub statement of one in S; for example the reassign statement is a sub statement of a if statement. In this case, MarkStatementsFollowingReAssign() have to mark all the statements in S, that either follows a reassign statement or following a statement that contains a reassign statement as a sub statement.

The second scenario is one special case of the first, where a reassignment appears inside a loop statement. In this case, in addition to the treatment as in the first, it is important to mark all the statements inside the body of the loop and also the loop statement as inappropriate for push operation. The loop statement has to be marked to avoid any replacement on the looping conditional expression.

3.3 Expression Simplifying Algorithm

TXL C Grammar overrides for operator precedence

The TXL provides its own grammar definitions for C language, supporting ANSI C and GNU extensions. For some reason, the binary expression defined in TXL C grammar doesn't facilitate for operator precedence. Hence an expression simplifying algorithm with this grammar is very difficult and inefficient. Therefore any required literals in the grammar are redefined to override the default, in order to support for operator precedence. The definition for binary_expression as defined in the TXL grammar is a unary_expression or a binary_expression followed by a binary_operator and a unary_expression. Parsing a binary expression "a + b * c – d" will construct the parse tree as ((((a) + b) * c) – d).

Due to this nested form of the binary_expression definition, a new grammar for conditional_expression is defined to override the TXL grammar. The new scheme defines as follows.

assignment_expression:
```
redefine assignment_expression
        [unary_expression_assign*] [conditional_expression]
end redefine
define unary_expression_assign
        [unary_expression] [assignment_operator]
end define
```

conditional_expression: A conditional_expression is defined as a sequence of logical_AND_expressions, associated by logical OR operator. logical_AND_expressions further defined to handle the precedence of other logical operators such as NOT, XOR, equals etc. Polynomial expressions in a program are identified as additive_expressions, which are the building blocks of logical expressions. The expression simplifying algorithm implemented in this project, only simplifies polynomial expressions and not any logical expressions. The further details on additive_expressions are discussed in the algorithm section below, while the complete definitions of the overridden grammar can be found in Appendix A.

Expression Simplifying algorithm

As discussed above, polynomial expressions are the building blocks of the logical expressions. A single polynomial expression appear in a C program, is considered as a single logical_AND_expression. Therefore any polynomial expression can be considered as a logical expression. A polynomial is identified as an additive_expression, which is defined as a multiplicative expression followed by a sequence of additive_operator and multiplicative_expression pairs.
Sequence of multiplicative_expressions are associated with additive_operators such as '+' and '-'. This grammar structure ensures lower precedence to additive operations by considering multiplicative_expressions as units in the additive expressions.

```
define additive_expression
        [multiplicative_expression]
[add_subtract_multiplicative_expression*]
end define

define add_subtract_multiplicative_expression
        [additive_operator] [multiplicative_expression]
end define

define additive_operator
        '+ | '-
end define
```

For example, `3*a+4*b+c+(d+e)`is an additive expression, constructed with `3*a`, `4*b`, `c` and `(d+3)` multiplicative_expressions. Apart from the first multiplicative expression 3 * a, all others have an additive operator preceding it. add_subtract_multiplicative_expressions are constructed from additive operators and multiplicative expressions. Therefore the polynomial `3*a+4*b+c+(d+e`) is parsed into `3*4` as the first multiplicative_expression followed by add_subtract_multiplicative_expressions `‘+ 4* b’`, `‘+ c’` and `‘+ (d  + e)’`.

cast_expression: is a unary_expression having optional type casting operators preceding it. For example `(3 + 5)` is a unary_expression and `‘(float)(3+5)’` is identified as a cast_expression.

Similarly a multiplicative_expression is defined as a cast operation followed by zero or more multiply_divide_cast_expressions, where a multiply_divide_cast_expression is a multiplicative operator (`*, /` and `%`) followed by a cast_expression.

The definitions are as follows.

```
define multiplicative_expression
        [cast_expression] [multiply_divide_cast_expression*]
end define

define multiply_divide_cast_expression
        [multiplicative_operator] [cast_expression]
end define

define multiplicative_operator
        '* | '/ | '%
end define

define cast_expression
        [cast_operator*] [unary_expression]
end define
```

Algorithm

```
SIMPLIFY_EXPRESSIONS(Program P)
begin
Do
      Program NewP = P
      EliminateParanthesis(NewP)
      ResolveNumericalMul(NewP)
      CoefficientsFirst(NewP)
      ResolveNumericalAdd(NewP)
      AddUnitCoefficient(NewP)
      AddZeroAdditive(NewP)
      AddMulTerms(NewP)
      EliminateUnitCoefficients(NewP)
      EliminateZeroTerms(NewP)
While (P not equal to NewP)
```

`EliminateParanthesis(Program P):`

Eliminating unnecessary parentheses is vital for a better expression simplification. For example an expression like `(i+j)+i` required the parentheses being eliminated to get a simple additive_expression i`+j+i` and to further simplify it to `2*i+j`.
This function eliminates any unnecessary parameters that exist in or introduced to the additive_expressions during the simplifying process. Parentheses around a unary_expression such as a number or an identifier, parentheses around multiplicative_expressions, parentheses around an additive_expression within another additive_expression are few examples of unnecessary parentheses those are eliminated here.

`ResolveNumericalMul(Program P):`

This function finds numerical unary_expressions within a multiplicative_expression and resolves them to get a single coefficient to that multiplicative_expression. For an example, a multiplicative_expression `3*i*4*j*2` will get simplified into `24*i*j`. The implementation of this function perfectly handles multiply operations; but handling divisions properly required the grammar to define with more precedence levels such a way division gets higher priority than multiplication, which is not handled here. However the grammar defined above, can be easily extended to support these features.

`CoefficientsFirst (Program P):`

This function brings the single numerical factor of a multiplicative expression as the first unary_expression of the multiplicative_expression. (There can be only one numerical factor at most, since `ResolveNumericalMul()` function resolve the numerical multiplications fist). This function is being used in adding equal multiplicative expressions together in `AddMulTerms()` function.

`ResolveNumericalAdd(Program P):`

This function adds together any multiplicative_expressions in an additive_expression, which are just numbers. For example in an additive expression `3+4*i+7+j`, `3` and `7` are two multiplicative_expressions those can be add together to obtain an additive_expression `10+4*i+j`.

`AddUnitCoefficient(Program P):`
In order to facilitate writing generalized rules for further simplifications, it is advantages to have a coefficient to all the multiplicative expressions. In an instance where an additive_expression `4*i*j+i*j`, having two multiplicative_expressions, it is convenient to have a coefficient to each multiplicative_expression. Once it is done the expression above is turned into an equivalent expression `4*i*j+1*i*j.` Then those two multiplicative_expressions are add together and obtain `5*i*j`. This function adds a unit coefficient to all multiplicative expressions to those don't have a coefficient.

`AddZeroAdditive (Program P):`
In these grammar definitions for expression with operator precedence, the first multiplicative_expression of an additive expression is identified differently than the following ones. The first one is identified as just a multiplicative_expression and the following ones are add_subtract_multiplicative_expressions, which are pairs of additive operator and multiplicative_expression. The `AddZeroAdditive()` adds a zero as the first multiplicative in the additive expressions to facilitate writing more generalized rules. Hence all effective multiplicative terms of an additive expression are shifted right and identified as add_subtract_multiplicative_expressions.

`AddMulTerms(program P):`
This function add together equal multiplicative terms; for example multiplicative terms in an additive expression `3*i*j+4*j*i,` are add together to get `7*i*j`. In order to get this additions done, `AddMulTerms()` required a numerical coefficient to each of the multiplicative terms considered to add together. The previous function calls ensure that, there is exactly one numerical factor in each multiplicative terms, by way of arithmetic resolution of all numerical factors and bring it to the first position of the multiplicative expression. If there is no numerical factor in a multiplicative term, `AddUnitCoefficient()` introduces a numerical multiplicative factor of 1 in the first position of each multiplicative expression, if it doesn't have a numerical factor in it.

This function deconstructs the multiplicative expression into the numerical factor and the list of non-numerical factors. To check for equality, this function check the length of the non-numerical factor list of two multiplicative terms, and also check the existence of each factor in one term in the other. This ensures that two multiplicative expressions with non-numerical factors in different order are identified as equal terms. An expression `i*j*k` and another expression `k*j*i` are successfully add together and produce `2*i*j*k`.

`EliminateUnitCoefficients(program P):`
Eliminate any unit coefficient in multiplicative terms, those either exist in the original program or introduced by the algorithm. An expression `1*i*j` will get replaced with `i * j`.

`EliminateZeroTerms(program P):`
Similar to above, any zero elements in additive expressions, those were either introduced by the algorithm or exist in the original program are eliminated. An additive expression `0+5*j+k` will get replace with `5*j+k.`

Few examples of expressions and its simplified version by this algorithm are follows.

```
   o  (3+5)+7*(3+5)+(7)+6+7+(7+8)      =>    99
   o  2+2*i*j+3+4*j*i+j+20*i+3*j+k*j   =>
                              5+6*i*j+4*j+20*i+k*j
```

3.4 Eliminating Killed Assignments

The definitions of *kill(n)*, *effectivekill(n)* and *redundant assignment* defined in the Definitions section of this report are better understood with the following examples.

```
void main(){
1    iVariant = 0;
2    intern = 3;
3    printf("%d", intern);
4    iVariant = 3;
5    printf("%d", iVariant);
6    intern = iVariant + intern;
7    printf("%d", intern);
}
```

Figure 3-12

In the sample program shown in Figure 3-12, line 1 assigns 0 to `iVariant`, and line 4 assigns a new value 3 to `iVariant`. Further the variable `iVariant` is not being reference anywhere in *Succ(2) – Succ(4)*. Since the value assigned at line 2 is not being used in down the program, the assignment statement at line 2 is called as a *redundant assignment* and the assignment statement at line 4, which overrides the previous assignment is called as a kill statement to line 2, that is *kill(2)*. Since the kill statement at line 4 makes the previous assignment at line 4 redundant, *kill(2)* at line 4 is also an *effectivekill(2)*.

In a program, any redundant assignments can be eliminated without loosing the semantics of the original program.

Conditional kill statements

A kill statement in a sequence of statements may cause an assignment redundant, within a single block of statements. However a kill statement in a conditional block can't be an effectivekill, if any statement outside the conditional block has any reference to the assignment variable. This is due to the conditional nature of the kill statement and only at runtime it can be determined whether it is an effective kill or not.

Consider the following program,

```
1  void main(){
2    int i, j, k;
3    i = 0;
4    j = 1;
5    if(j < 0 ){
6         i = 10;
7         j = 5;
8         printf(“Conditional: %d, %d”, i, j);
9    }

10   printf(“Outter: %d”, i);
11 }
```

Figure 3-13

Here, at line 3, `i` is assigned to `0`, and conditionally killed at line 6. Though there is no statement between the assignment and the kill statement, have a reference to variable `i`, still assignment at line 3 is not made redundant, because of line 10 which has a reference to `i` outside the conditional body. On the other hand, the assignment to `j` in line 4 is killed at line 7 conditionally, and also none in *Succ(4) – Succ(7)* has a reference to `j`. In addition, statements in *Succ(5)* but outside the conditional block, also have no reference to `j`. Therefore the kill statement *kill(4)* at line 7 is also an *effectivekill(4)*.

However, a kill statement at *m* to a conditional assignment at *n*, is an *effectivekill(n)* if it is not being referenced by any other statements in *Succ(n) – Succ(m)*. This is better understood with the following example.

```
1  void main(){
2     int i, j, k;
3     scanf(“%d”, j);
4     if(j < 0 ){
5           i = j;
6           printf(“Conditional:%d”, j);
7     }
8     i = 2 * j;
9     printf(“Outter: %d”, i);
10 }
```

Figure 3-14

In the program shown in figure 3-14, line 8 is a kill statement to the assignment at line 5, that is line 8 is a *kill(5)*. Since there is no reference to the assignment variable `i` in *Succ(5) - Succ(8)*, assignment statement at line 8 is also an *effectivekill(5)*; also assignment at line 5 is a *redundant assignment*.

The redundant statement elimination algorithm for programs with conditional statements can be applied for programs with loop statements as well. An assignment statement at *m* inside the body of a loop effectively kills a previous assignment outside the loop at *n*, if and only if there is no reference to the assignment variable in *Succ(n) – Succ(m)* and also in the statements following while loop.

Algorithm

This algorithm examines each of the assignment statement at *n* for an effective kill statement in *Succ(n)*. In the first phase, the algorithm figure out all redundant assignments and corresponding effective kill statements in a single block of statement and eliminate the redundant assignments. Once it is done, the algorithm mark any assignment statements left and also doesn’t have a corresponding effective kill statement; this is done considering nested block statements as well. Finally all the unmarked assignments are removed from the program, and all marked assignments are unmarked.

```
EliminateRedudentAssignments (program P)
begin
Do
     Program NewP = P
     For each block of statements block in NewP
          SimpleEliminate(block)
          MarkBlockLevelDepAssignments(block)
          MarkInternalBlockDepAssignments(block)
          MarkOutofBlockDepAssignments(block)

     DeleteAllUnmarkedAssignments(NewP)
While (NewP not equal to P)
end
```

`SimpleEliminate(block):`

This rule, apply on the statements in a single block; each assignment in the block having an effective kill statement in the block are eliminated. An effective kill statement in the block is either an effectivekill in the same nested level of the block or a statement in the same nested level, include an effectivekill statement as a sub-statement of it.

Consider the example in Figure 3-15. `SimpleEliminate()` eliminates the assignment to `b` at line 5, but preserves the assignment to `a` in line 4, as it is being used in the body of the if statement.

```
1 void main(){
2    int a, b, N;
3    scanf("%d", N)
4    a = 0;
5    b = 0;
6    if(N > 0){
7         printf("%d", a);
8    }
9    b = 2 * a;
10   printf("%d", b);
11 }
```

Figure 3-15

`MarkBlockLevelDepAssignments(block b):`

This function simply mark any assignment statement in block b (say at line *n*), if there is any reference to the assignment variable in succ(n) – succ(m); here m is the lowest position of an effective kill statement to n if there is one, otherwise m is the end of the program. Those assignments are not redundant assignments, hence marked as <non-redundant>. If a statement is identified as non-redundant it is so, and can't be eliminated from the program. However assignments those are not identified as non-redundant can't be concluded as redundant until further examinations by the other functions also fail to identify them as non-redundant.

`MarkInternalBlockDepAssignments(block b):`

This function examines any assignment statement in an outer block that have a kill statement in an inner block. If the kill statement is not an effective kill then the assignment statement is marked as non-redundant. In the following example this function mark the assignment to `b` at line 5 as non-redundant assignment, as it is referenced in line 11, outside the conditional if block which reassigns b. However assignment to a in line 4 is effectively killed by the reassignment in line 7, since the reference to a is also in the same conditional block as the reassignment. Therefore, this function left the assignment statement at line 4 unmarked.

```
1 main(){
2     int a, b, N;
3     scanf(“%d”, N)
4     a = 0;
5     b = 0;
6     if(N > 0){
7          a = 1;
8          b = 1;
9          printf(“%d, %d”,a, b);
10    }
11    printf(“%d”, b);
12 }
```

`MarkOutofBlockDepAssignments(block b):`
This function examines the assignments statements in an inner block for dependency with succeeding statements in the outer block. If the assignment statement examining is effectively killed by a succeeding statement in the outer block, it is considered as a redundant assignment and left for deletion at the end; otherwise marked as non-redundant. In the following program, this function identifies assignment to b in the inner block at line 9 as non-redundant, because it is referenced in line 11 in the outer block. But assignment to a at line 8 is left unmarked, as it is considered as a redundant statement.

```
1 main(){
2     int a, b, N;
3     scanf(“%d”, N)
4     a = 0;
5     b = 0;
6     if(N > 0){
7          printf(“%d, %d”,a, b);
8          a = 1;
9          b = 1;
10    }
11    printf(“%d”, b);
12 }
```

Finally all the unmarked assignment statements are eliminated from the program, and the marked statements are converted back to unmarked statements.

Similar algorithms were implemented to handle blocks of loop statements very similar to as described above.

3.5 Pointer Elimination

The amorphous slicer implemented in this project, handles slicing in the presence of pointers by eliminating the pointer references by replacing the dereferences with the target objects, and apply the previously discussed slicing algorithms to it.

Algorithm

The pointer elimination algorithm described here, handles pointers of static objects, pointers to pointer, pointer reassignments etc. The algorithm discussed here deals with each address assignment statements in the program from bottom to top, replacing all it's dereferencing in the succeeding statements. This is done in a bottom to top fashion to ensure address reassignments are handled properly.

In case of pointers to objects (against to pointers to pointer), the address assignments can be eliminated once all the succeeding dereferences are replaced. However an address assignment to a pointer to pointer may make things complicated and required a sophisticated approach. In order to handle this properly, this algorithm mark the dereferenced address assignments instead of eliminating them, therefore those assignment statements are available for further processing later if required. If a processed (marked) address assign statement is affected by a address dereference replacement by a preceding address assignment, it will be unmarked to allow being processed again.

```
EliminatePointers(SequenceOfStatements S){
     while(address assignment found in S) do

          for each address assign stmt s E S do
               M <- Succeeding statements to s
               if(M has Address Assignment)
                    EliminatePointers(M)

               deconstruct s = 'p = &a' into
                    pointer = p
                    target = a
               for each statement s' in M do
                    if(s' reference to p){
                         unmark(s')

                    replace *pointer with a

               mark(s)
     end while
     remove all marked statements in S

end
```

Figure 3-16

Since the algorithm is recursively called on sequence of statements until there is no address assignment in the succeeding statements, pointer dereferences are eliminated from bottom to top. In the sample program in Figure 3-17, the address assignment `*r = &b;` at line 6 is analysed first, replacing dereferencing `*(*r)` in line 8 with `b`, therefore line 8 will get replaced with `b = 78;` and line 6 is marked. Now assignment statement `r = &p;` at line 5 is analysed and all dereferences to `r`, `*r` will get replaced with `p`; line 6 will be replaced with `p = &b`, and line 6 get unmarked to allow for a second processing; line 5 is marked, as it has been processed. Now the unmarked addresses assign statement at line 6 is the bottom most address assignment to be processes; since line 6 is already replaced with `p = &b`;, all the dereferences to `p`, `*p` are replaced with `b` at line 7 and line 9. Finally address assignment at line 4 is processed, replacing none as address assignment to p at line 4 is overridden again at line 6. Finally all the marked statements (address assignment statements) are eliminated. The expected output is similar to the one as shown in Figure 3-18.

```
1  void main(){
2     int *p, **r;
3     int a; int b;

4     p = &a;
5     r = &p;

6     *r = &b;
7     *p = 60;

8     **r = 78;

9     printf("%i",*p);
10 }
```

Figure 3-17

```
void main(){
      int a; int b;

      p = &b;
      b = 60;

      b = 78;

      printf("%i",b);
 }
```

Figure 3-18

Pointer elimination in the presence of conditional statements

The pointer elimination algorithm works in the bottom up fashion as discussed before for a single block of statements. However any conditional address assignments leads to the necessity to protect the dereferencing statements with equivalent condition when dereferences are replaced.

In the following program, address assignments to pointers y and z are conditional. Therefore, replacing dereferences to y and z, required those statements having references to be protected with equivalent conditions. Address assignment at line 17 doesn't have any dereferences succeeding it; therefore it is marked as processed. Then address assignments at lines 14 & 15 are processed, replacing dereferences to z with either y or w guarded with equivalent conditional statement. Now line 17 is processed again, followed by assignments in line 11 and 12. A step by step transformation of the original program is listed in figure 3-19.

```
main(){
      int a, b, s, u, x;
      int *w, *y, **z;

      s =1;
      a = 2;
      b = 3;

      scanf("%d %d", &x, &u);

      if(x) y = &a;
      else    y = &b;

      if(u) z = &y;
      else z = &w;

      w = &s;
      **z = 4;

      printf("%d", s);
}
```

```
main(){
      int a, b, s, u, x;
      int *w, *y;

      s =1;
      a = 2;
      b = 3;

      scanf("%d %d", &x, &u);

     if(x) y = &a;
     else    y = &b;


     w = &s;
       if(u) *y = 4;
       else *w = 4;

     printf("%d", s);
}
```

```
main(){
      int a, b, s, u, x;

      s =1;
      a = 2;
      b = 3;

      scanf("%d %d", &x, &u);

       if(u){
           if(x) a = 4;
           else b = 4;
       }
       else s = 4;

     printf("%d", s);
}
```

Figure 3-19

Applying slicing on the resulting program as show in figure 3-19, the output will be as shown in figure 3-20.

```
main(){
      int s, u, x;
      s =1;
      scanf("%d %d", &x, &u);
       if(u){
       }
       else s = 4;

     printf("%d", s);
}
```

Figure 3-20

4 EVALUATION

4.1 Evaluation Scheme

The amorphous slicer developed as a part of this project, had been evaluated for both its correctness and slicing factor. The correctness of the semantics of generated slices compared with the semantics of the original program is tested by compiling and executing the original program and the slice; the outputs from both are compared for same behaviors.

An amorphous slicer is expected to generate the thinnest possible slice for the chosen slicing criterion. A manual walk through on the original program and slice is necessary to ensure that all possible eliminations are executed while amorphous slicing. However, in this project only a certain amorphous slicing scenarios are handled, such as push transformation, pointer elimination, elimination of redundant assignments and expression simplification. Further the aim of the implementation here is proving the correctness of the concepts and algorithms; and only few chosen syntaxes out of those falling in one concept are implemented. For example to explain the behavior of the algorithms in the presence of loop statements, a while loop is considered for implementation. There are many other loop statements such as do while statements, for statements etc. which doesn't considered for implementation as they behave very similar to while statements.

4.2 Evaluation of functional precision

Test run -1
Figure 4-1 lists the input program, which computes the Fibonacci numbers for integers less than N, and also computes the largest prime less than N. The amorphous slices generated for each of these functions are listed in Figure 4-2 and 4-3.

```
/* Computing the first N fibonacci numbers and the largest prime
smaller than N */

int main(void) {
    int i;        /* The index of fibonacci number to be printed
next */
    int current;  /* The value of the (i)th fibonacci number */
    int next;     /* The value of the (i+1)th fibonacci number */
    int twoaway;  /* The value of the (i+2)th fibonacci number */
    int N;

    int curr_prime;

    N = 30;
    curr_prime = -1;

    printf("Computing first %d Fibonacci numbers. ");
    if (N <= 0)
          printf("The number should be positive.\n");
    else {
          printf("\n\n\tI \t Fibonacci(I)
\n\t=====================\n");
          next = 1;
19          current = 1;
20          i = 1;
21          while( i <= N){
22                int j, mod;
23                int is_prime;

24                printf("\t%d \t   %d\n", i, current);
25                twoaway = current+next;
26                current = next;
27                next    = twoaway;


                  /*computing the current largest prime */
28                is_prime = 1; //true
29                if(i > 1){
30                      j = 2;
31                      while( j < i){
32                            mod = i % j;
33                            if(mod == 0){
34                                  is_prime = 0;
35                            }
36                            j = j + 1;
37                      }
38
39                }
40                else{
41                      is_prime = 0;
42                }

43                if(is_prime){
44                      curr_prime = i;
45                }

46                i = i +1;
47          }
48
49          printf("\n\t Biggest prime less than %d is %d.\n", N,
curr_prime);
50    }
51 }
```

Figure 4-1

Amorphous slice for the program listed in Figure 4-1, for the slicing criterion, <49, { `curr_prime`}>, computing the biggest prime. In this slice the push transformations are applied to all possible places, without loosing the semantics of the original program. Further the static slicer perfectly eliminates the irrelevant codes.

```
#include <stdio.h>

int main (void) {
    int i;
    int curr_prime;
    curr_prime = - 1;
    if (30 <= 0) printf ("The number should be positive.\n");
    else {
        i = 1;
        while (i <= 30) {
            int j;
            int is_prime;
            is_prime = 1;
            if (i > 1) {
                j = 2;
                while (j < i) {
                    if (i % j == 0) {
                        is_prime = 0;
                    }
                    j = j + 1;
                }
            }
            else {
                is_prime = 0;
            }
            if (is_prime) {
                curr_prime = i;
            }
            i = i + 1;
        }
        printf ("\n\t Biggest prime less than %d is %d.\n", 30,
curr_prime);
    }
}
```

Figure 4-2

Amorphous slice for the program listed in Figure 4-1, for the slicing criterion, <24, {i, current}>, computing Fibonacci numbers for first N integers.

```
#include <stdio.h>
int main (void) {
    int i;
    int current;
    int next;
    int twoaway;
    if (30 <= 0) printf ("The number should be positive.\n");
    else {
        next = 1;
        current = 1;
        i = 1;
        while (i <= 30) {
            printf ("\t%d \t   %d\n", i, current);
            twoaway = current + next;
            current = next;
            next = twoaway;
            i = i + 1;
        }
    }
}
```

Figure 4-3

Test run – 2

The program listed in Figure 4-4, extracted from the paper *“Program Slicing in the presence of Pointers”* by James.R.Lyle and David Binkley. The sample run here explains amorphous slicing with pointers.

```
#include <stdio.h>
void main() {
    int a, b, s, u, x;
    int *w, *y, **z;

    s = 1;
    a = 2;
    b = 3;
    scanf("%d %d", &x, &u);

    if(x){
          y = &a;
    } else{
          y = &b;
    }

    if(u){
          z = &y;
    } else{
          z = &w;
    }
    w = &s;
    **z = 4;
    <mark> printf("s %d \n", s); </mark>
 }
```

Figure 4-4

The amorphous sliced produced on the program listed in Figure 4-4 for the slicing criterion <23, {s}> is shown in Figure 4-5. In this slice, line 8 is not included in the slice; which is incorrect. This is because, a function calls with parameters by references are not handled in the current implementation as assignments to those parameters. The pointers are eliminated first and push transformation with conditional statements, and static slicing are applied perfectly.

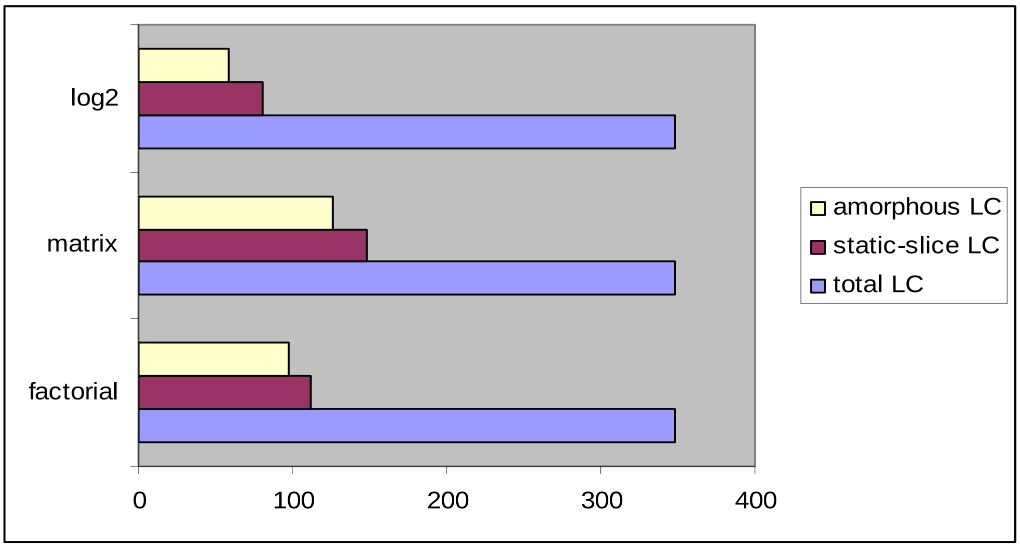
```
#include <stdio.h>

void main () {
    int s, u;
    s = 1;
    if (! u) {
        s = 4;
    }
    printf ("s %d \n", s);
}
```

Figure 4-5

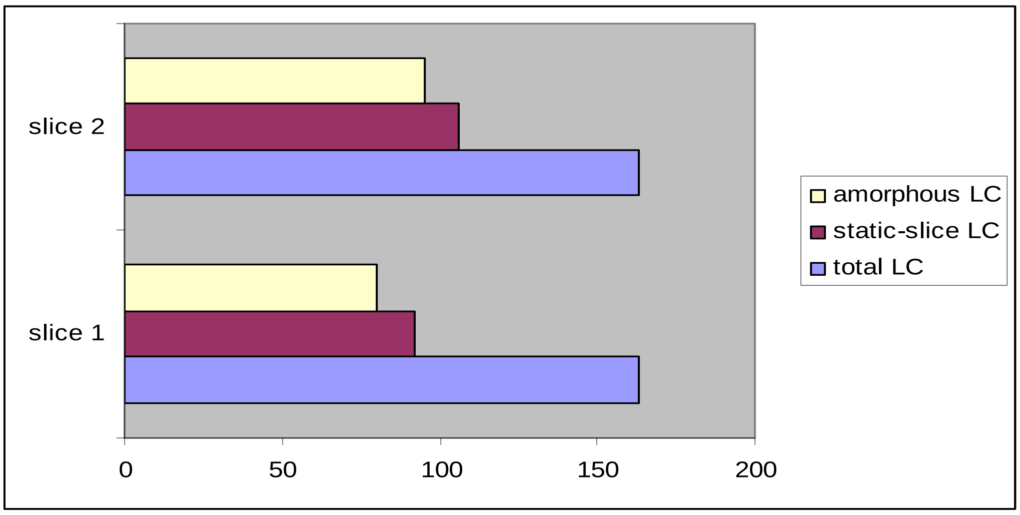
4.3 Efficiency of Amorphous slicer compared with static slicer

Following are some statistics of few slicing run with static slicer and amorphous slicer compared the results with the line count of the input program.

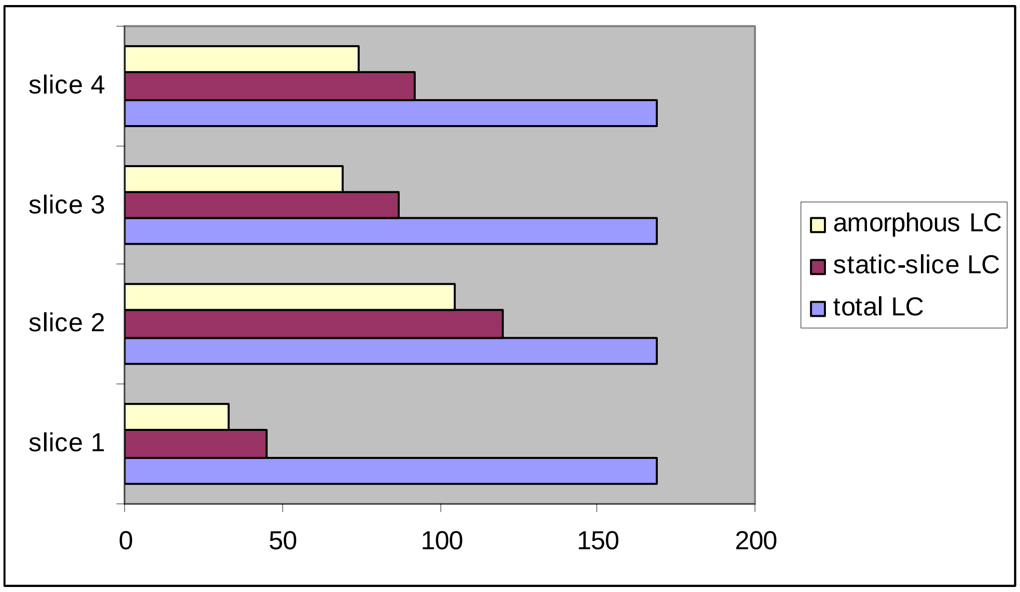
I. math.c: a utility that provides functionalities for matrix computations, log2, and factorial.



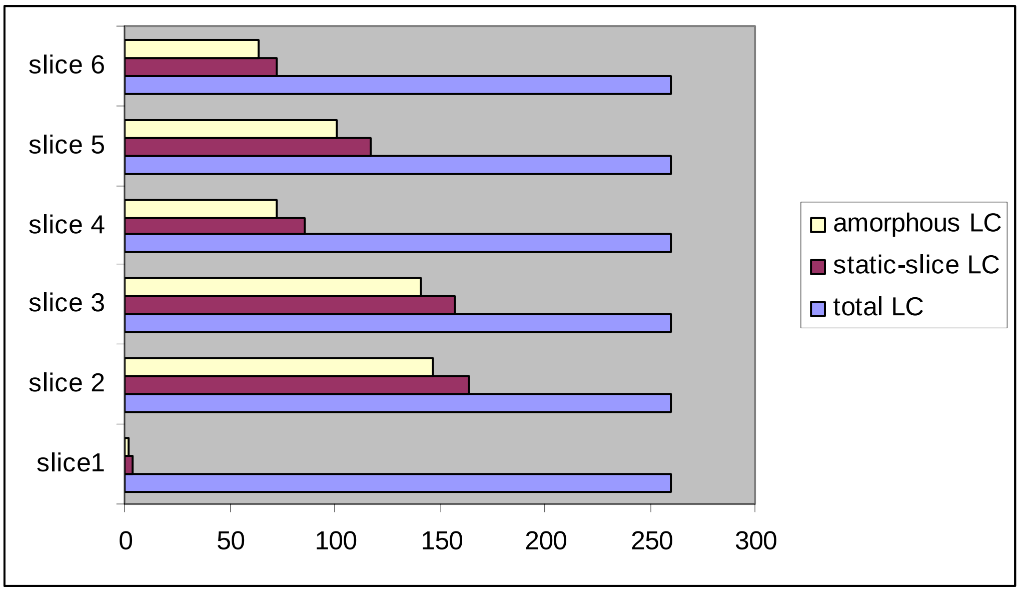
II. Program for the eigenvalue problem with a combination of the bisection method and the Numerov algorithm is sliced on different variables.



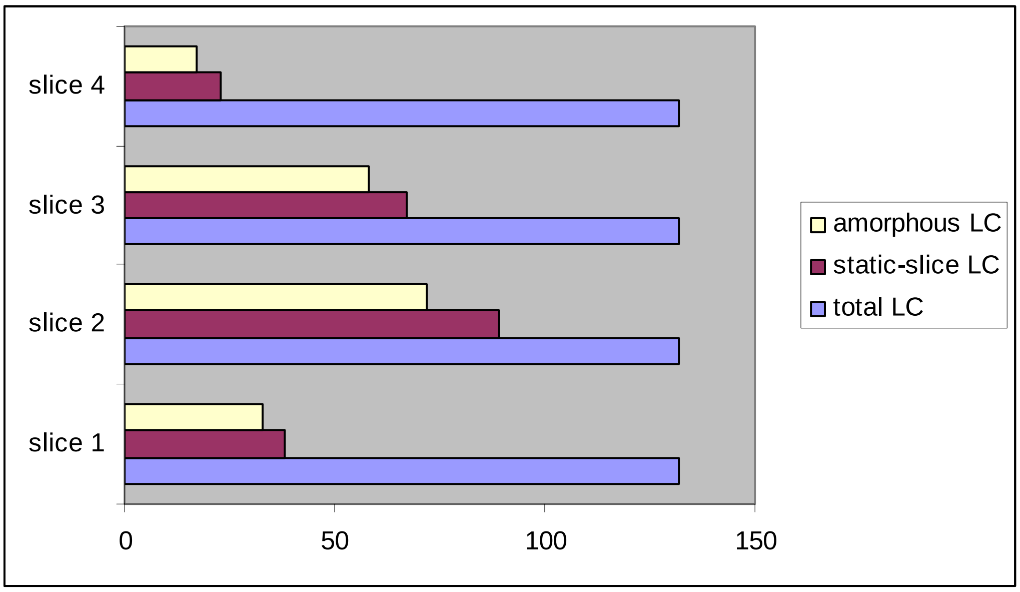
III. Slicing on program computing chemical bond length on NaCl on different variables



IV. Slicing on program constructing the singular value decomposition of any matrix, as implemented in Numerical Recipes in C.



V. Slicing on a program that solves time dependent temperature field around a nuclear waste rod in a two dimensional model, as implemented in www.physics.unlv.edu. Slicing applied on different variables in the program repeatedly.



The summary of the slicing results shown above is tabulated here.

total Line Count	**static-slice Line Count**	**amorphous slice Line Count**
348	112	98
348	148	126
348	80	58
163	92	80
163	106	95
169	45	33
169	120	105
169	87	69
169	92	74
260	4	2
260	164	147
260	157	141
260	86	72
260	117	101
260	72	64

The static slicer produces slices of line count 41% of the input program on average, where the amorphous slicer on average produces 35% of the input program. Amorphous slices are on average 85% of the static slices.

REFERENCES

[1] James R.Lyle, David Binkley .*Program Slicing in the presence of Pointers*, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology.

[2] Mark Harman and Sebastian Danicic. *Amorphous Program Slicing*. Project project, School of Computing, University of North London, Eden Grove, London, N7 8DB, UK.

[3] Mark Harman, Lin Hu, Malcolm Munro, Xingyuan Zhang, Dave Binkley, Sebastian Danicic, Mohammed Daoudi, Lahcem Ouarbya. *Syntax Directed Amorphous Slicing*.

[4] Mark Harman, Yoga Sivagurunathan and Sebastian Danicic. *Analysis of Dynamic Memory Access using Amorphous Slicing*. Department of Mathematical and Computing Sciences, Goldsmiths College, University of London, London.

[5] Jeff Russell. *Program Slicing Literature Survey*. December 2001.

[6] Todd M. Austin and Gurindar S. Sohi. *Dynamic Dependency Analysis of Ordinary Programs*. Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton street, Madison, WI 53706.

[7] Laurent Théry. *Simplifying Polynomial expressions in a proof assistant.* Théme 2 – Génie logiciel et calcul symbolique, Project Lemme, Juin 2005.

[8] James R. Cordy. *Source Transformation, Analysis and Generation in TXL*. School of Computing, Queen'sUniversity, Kingston,Canada.

[9] James R. Cordy, Ian H. Carmichael and Russell Halliday. *The TXL Programming Language*, Version 10.4, January 2005.

[10] www.txl.ca as on August 30, 2007.

[11] www.grammatech.com as on August 30, 2007.

APPENDIX -A

C Grammer as Defined in TXL

```
% TXL Base Grammar for ANSI and K+R C

% Author: A Malton, University of Waterloo, Feb 2002
% Based on the TXL release ANSI C 7.0 grammar by
% J.R. Cordy, Queen's University, January 1994

% This is a TXL base grammar for C, including
%    * ANSI features (e.g. literal string concatenation)
%    * old-style (K&R) function parameters
%
% but excluding:
%    * preprocessor directives
%    * comments
%    * C++ features
%    * string literal

% C and C++ style comments are accepted but not parsed.

% MODIFICATION LOG:
%
% Corrected formatting cues to fix problems with lack of spacing -
JRC 30.5.04
% Added handling of preprocessor directives - JRC 8.1.03
%

% Comment out this line to disallow preprocessor directives
#define PREPROCESSOR
%

% Lexical properties of C

#pragma -idchars '$' -width 160

comments
    //
    /* */
end comments

tokens
    hex                 "0[xX][\dAaBbCcDdEeFf]+[LUlu]*"
    dotfloat            ".\d+([eE][+-]?\d+)?[FLfl]?"
    float         "\d+.\d*([eE][+-]?\d+)?[FLfl]?"
            |     "\d+(.\d*)?[eE][+-]?\d+[FLfl]?"
            |     "\d+(.\d*)?([eE][+-]?\d+)?[FLfl]"
    long                  "\d+[LUlu]+"
#ifdef PREPROCESSOR
    id            |     "\#\i+"
#endif
end tokens
```

```
keys
        auto            double          int             struct
        break           else            long            switch
        case            enum            register        typedef
        char            extern          return          union
        const           float           short           unsigned
        continue        for             signed          void
        default         goto            sizeof          volatile
        do              if              static          while
#ifdef PREPROCESSOR
       '#define '#else '#endif '#if '#ifdef '#ifndef '#include '#line
'#undef '#indent '#LINK
#endif
end keys

compounds
        ->      ++      --      <<      >>      <=      >=      ==
!=
        &&      ||      *=      /=      '%=     +=      -=
        <<=     >>=     &=      ^=      |=
end compounds

define C_compilation_unit
    [repeat declaration_or_function_definition]
end define

% Constants

define constant
    [number]
|   [float]
|   [hex]
|   [long]
|   [SP] [dotfloat]             % TXL doesn't defaultly space
before .
|   [charlit]                     % "single" character constant
|   [string]
end define

define string
    [repeat stringlit+]           % Includes implicit concatenation
end define

% Expressions
define expression
    [list assignment_expression+]
end define

define constant_expression
    [conditional_expression]
end define

define assignment_expression
    [conditional_expression] [opt assignment_operation]
end define

define assignment_operation
```

```
    [assignment_operator] [assignment_expression]
end define

define assignment_operator
    = | *= | /= | '%= | += | -= | >>= | <<= | &= | ^= | '|=
end define

define conditional_expression
    [binary_expression] [opt conditional_operation]
end define

define conditional_operation
    ? [expression] : [conditional_expression]
end define

define binary_expression
    [unary_expression]
|   [binary_expression] [binary_operator] [unary_expression]
end define

define binary_operator
     +  | -  | *  | / | '%
|   ==  | != | <  | > | <= | >=
|   '|| | && | '| | ^ | &
|   <<  | >>
end define

define unary_expression
    [postfix_expression]
|   [unary_operator] [SPOFF] [unary_expression] [SPON]
|   ( [type_name] ) [unary_expression]
|   [sizeof_expression]
end define

define sizeof_expression
    'sizeof ( [type_name] )
|   'sizeof [unary_expression]
end define

define unary_operator
    * | & | + | - | ! | ~ | ++ | --
end define

% A postfix expression might refer to some part
% of a named thing (e.g. x [4]. a. b)
% or a computed thing (e.g. x [4] (i). x. y)
% or a literal thing (e.g. "foobar" [3]).

% This grammar assigns uid reference markup to the first category
only.

define postfix_expression
    [reference]
|   [nonreference]
end define

define reference
```

```
    [reference_id]
|   [reference_expression]
end define

define reference_id
    [id]
end define

define reference_expression
    [unannotated_reference_base] [repeat postfix_extension]
end define

define unannotated_reference_base
    [reference_id]
|   [nonreferential_primary]
end define

define nonreference
    [nonreferential_primary] [repeat postfix_extension]
end define

define nonreferential_primary
    [constant]
|   [string]
|   '( [expression] ')
end define

define postfix_extension
    '[ [expression] ']
|   '( [opt expression] ')
|   '. [id]
|   '-> [id]
|   '++
|   '--
end define

% Declarations

% A declaration starts with a decl_specifiers and ends with
declarators.
% The decl_specifiers is a sequence of "declaration specifiers",
% each of which is either a type specifier (e.g. "int")
% or a qualifier of some kind (e.g. "extern").
% There can't be more than one type specifier.
% Here we ensure that there is either exactly one [type_specifier],
% or if the type was omitted, exactly one (empty) [opt
type_specifier].
% (Apart from any nested declarations in struct or union specs.)

define declaration
    [declaration_body] [semi]
#ifdef PREPROCESSOR
|   [preprocessor]
#endif
end define

define declaration_body
```

```
    [decl_specifiers] [list init_declarator+]
|   [enum_specifier]
|   [struct_or_union_specifier]
end define

define decl_specifiers
    [repeat decl_qualifier] [opt type_specifier] [repeat
decl_qualifier]
end define

% Structures

define struct_or_union_specifier
    [struct_or_union] [opt tagged_reference_id] {   [IN] [NL]
      [repeat member_declaration]                 [EX]
    }
|   [struct_or_union] [tagged_reference_id]
end define

% This kind of reference id is the kind used in struct... and enum...
definitions.
% It's in a different name space and often needs to be skipped.

define tagged_reference_id
    [reference_id]
end define

define member_declaration
    [decl_specifiers] [list member_declarator+] [semi]
#ifdef PREPROCESSOR
|   [preprocessor]
#endif
end define

define member_declarator
    [declarator] [opt bitfieldsize]
|   [bitfieldsize]
end define

define bitfieldsize
    ': [constant_expression]
end define

define decl_qualifier
    [sc_specifier]            % In ANSI C, not allowed for
member_declaraton.
|   [cv_qualifier]
|   [type_qualifier]
end define

define sc_specifier
    'auto
|   'register
|   'static
|   'extern
|   'typedef
end define
```

```
define type_specifier
    [simple_type_name]
|   [enum_specifier]
|   [struct_or_union_specifier]
end define

define type_qualifier
    'long
|   'short
|   'signed
|   'unsigned
end define

define simple_type_name
    'char
|   'int
|   'void
|   'float
|   'double
|   [type_id]
end define

define type_id
    [reference_id]
end define

define struct_or_union
    'struct | 'union
end define

define enum_specifier
    'enum [opt tagged_reference_id] { [list enumerator] }
|   'enum [tagged_reference_id]
end define

define enumerator
    [reference_id] [opt enumerator_value]
end define

define enumerator_value
    = [constant_expression]
end define

define init_declarator
    [declarator] [opt initialization]
end define

define declarator
    [repeat ptr_operator] [base_declarator] [SPON] [repeat
declarator_extension]
end define

define base_declarator
    [reference_id]
|   ( [declarator] )
end define
```

```
define declarator_extension
    [function_declarator_extension]
|   [array_declarator_extension]
end define

define function_declarator_extension
    ( [argument_declaration_list] ) [repeat cv_qualifier]
end define

define array_declarator_extension
    '[ [opt constant_expression] ']
end define

define ptr_operator
    * [repeat cv_qualifier] [SPOFF]
end define

define cv_qualifier
    'const
|   'volatile
end define

% For expressions mentioning types (e.g. casts and sizeof)

define type_name
    [type_specifiers] [opt abstract_declarator]
end define

% Can't be empty, and no more than one type.

define type_specifiers
    [repeat type_qualifier+] [opt type_specifier] [repeat
type_qualifier]
|   [type_specifier] [repeat type_qualifier]
end define

% This is a declarator which doesn't introduce a name, but is just
for mentioning types.

define abstract_declarator
    [repeat ptr_operator+] [repeat declarator_extension]
|   ( [abstract_declarator] ) [repeat declarator_extension]
end define

define argument_declaration_list
    [list argument_declaration]
end define

% An argument declaration is like a regular one except at most one
declarator, without initializer,
% is allowed.  This allows empty argument declaration, which has a
reasonable meaning in theory,
% but not in C.

define argument_declaration
    [decl_specifiers] [opt argument_declarator]
```

```
|   '...          % Only allowed last in a non-empty list, never
mind.
end define

define argument_declarator
    [declarator]
|   [abstract_declarator]
end define

define initialization
    = [initializer]
|   ( [constant_expression] )
end define

define initializer
    [expression]
|   [NL] { [IN] [list initializer] [opt ',] [EX] }
end define

% Statements
define statement
    [repeat label] [unlabeled_statement]
#ifdef PREPROCESSOR
|   [preprocessor]
#endif
end define

define label
    [label_id] ':
|   [EX][SP][SP] 'case [constant_expression] ': [IN] [NL]
|   [EX][SP][SP] 'default ': [IN] [NL]
end define

define label_id
    [id]
end define

define unlabeled_statement
    [expression_statement]
|   [if_statement]
|   [for_statement]
|   [while_statement]
|   [switch_statement]
|   [do_statement]
|   [null_statement]
|   [jump_statement]
|   [compound_statement]
end define

define null_statement
    [semi]
end define

define compound_statement
    { [IN] [NL]
      [compound_statement_body]
    } [opt ';] [NL]
```

```
end define

define compound_statement_body
    [repeat statement]        [EX]
|   [declaration]       % Prefer statements if possible.
    [compound_statement_body]
end define

define expression_statement
    [expression] [semi]
end define

define if_statement
    'if ( [expression] ) [statement] [opt else_statement]
end define

define switch_statement
    'switch ( [expression] ) [statement]
end define

define else_statement
    'else [statement]
end define

define while_statement
    'while '( [expression] ') [statement]
end define

define do_statement
    'do [statement] 'while ( [expression] ) [semi]
end define

define for_statement
    'for ( [opt expression] '; [opt expression] '; [opt expression] )
      [statement]
end define

define jump_statement
    'goto [label_id] [semi]
|   'continue [semi]
|   'break [semi]
|   'return [opt expression] [semi]
end define

% Top-Level

define declaration_or_function_definition
    [declaration]
|   [function_definition]
end define

define function_definition
    [NL] [decl_specifiers] [declarator] [opt KR_parameter_decls]
    [compound_statement] [NL]
end define

define KR_parameter_decls
```

```
    [NL] [IN] [repeat declaration+] [EX]
end define

define semi
    '; [NL]
end define

define program
    [C_compilation_unit]
end define

#ifdef PREPROCESSOR

% Parse preprocessor directive lines, but don't interpret them

define preprocessor
    '#define [id] '( [list id+] ')  [expression] [NL]
|   '#define [id]  [expression]  [NL]
|   [EX] '#else  [IN] [NL]
|   [EX] '#endif [NL] [NL]
|   [NL] '#if [expression] [IN] [NL]
|   [NL] '#ifdef [id] [IN] [NL]
|   [NL] '#ifndef [id] [IN] [NL]
|   '#ident [stringlit] [NL]
|   '#include [stringlit] [NL]
|   '#include < [SPOFF] [filepath] > [SPON] [NL]
|   '#line [integernumber] [opt stringlit] [NL]
|   '#undef [id] [NL]
|   '#LINK [stringlit] [NL]
end define

define filepath
    [file_id] [repeat slash_fileid]
end define

define file_id
    [id]
|   [key]
end define

define slash_fileid
    [slash] [file_id]
end define

define slash
    '/ | '\ | '. | ':
end define

#endif
```

Extensions to C Grammar

```
% add marked_statement also a valid statement; by redefine
% 'statement' and adding the old definitions.
redefine statement
      [marked_statement]
      | ...
end redefine

% defining marked_statement as a statement enclosed in xml tag.
define marked_statement
      [xml_tag] [statement] [xml_end]
end define

define xml_tag
      < [SPOFF] [id] > [SPON]
end define

define xml_end
      < [SPOFF] / [id] > [SPON]
end define

redefine reference
        ...
    |   [marked_reference]
end redefine

define marked_reference
        [id]
end define
```

Overrides to C Grammar

```
% When precedence is needed in C transformations,
% include these overrides following the C grammar

redefine assignment_expression
        [unary_expression_assign*] [conditional_expression]
end redefine

define unary_expression_assign
        [unary_expression] [assignment_operator]
end define

redefine conditional_expression
        [logical_OR_expression] [conditional_operation?]
end redefine

define logical_OR_expression
        [logical_AND_expression] [OR_logical_AND_expression*]
end define

define OR_logical_AND_expression
        '|| [logical_AND_expression]
end define
```

```
define logical_AND_expression
        [inclusive_OR_expression] [AND_inclusive_OR_expression*]
end define

define AND_inclusive_OR_expression
        '&& [inclusive_OR_expression]
end define

define inclusive_OR_expression
        [exclusive_OR_expression] [OR_exclusive_OR_expression*]
end define

define OR_exclusive_OR_expression
        '| [exclusive_OR_expression]
end define

define exclusive_OR_expression
        [AND_expression] [exclusive_OR_AND_expression*]
end define

define exclusive_OR_AND_expression
        '^ [AND_expression]
end define

define AND_expression
        [equality_expression] [AND_equality_expression*]
end define

define AND_equality_expression
        '& [equality_expression]
end define

define equality_expression
        [relational_expression] [equality_relational_expression*]
end define

define equality_relational_expression
        [equality_operator] [relational_expression]
end define

define equality_operator
        '== | '!=
end define

define relational_expression
        [shift_expression] [relational_shift_expression*]
end define

define relational_shift_expression
        [relational_operator] [shift_expression]
end define

define relational_operator
        '< | '> | '<= | '>=
end define
```

```
define shift_expression
        [additive_expression] [shift_additive_expression*]
end define

define shift_additive_expression
        [shift_operator] [additive_expression]
end define

define shift_operator
        '<< | '>>
end define

define additive_expression
        [multiplicative_expression]
[add_subtract_multiplicative_expression*]
end define

define add_subtract_multiplicative_expression
        [additive_operator] [multiplicative_expression]
end define

define additive_operator
        '+ | '-
end define

define multiplicative_expression
        [cast_expression] [multipy_divide_cast_expression*]
end define

define multipy_divide_cast_expression
        [multiplicative_operator] [cast_expression]
end define

define multiplicative_operator
        '* | '/ | '%
end define

define cast_expression
        [cast_operator*] [unary_expression]
end define

define cast_operator
        '( [type_name] ')
end define

redefine unary_expression
        [pre_increment_operator*] [sub_unary_expression]
end redefine

define pre_increment_operator
        '++ | '-- | 'sizeof
end define

define sub_unary_expression
        [postfix_expression]
    |   'sizeof '( [type_name] ')
    |   [unary_operator] [cast_expression]
```

```
end define

redefine unary_operator
        '& | '* | '+ | '- | '~ | '!
end redefine
```

APPENDIX -B

slicer.sh
```
#!/bin/sh
txl $1 main.txl > parse_1.c
txl parse_1.c  main2.txl > out.c
```

main.txl
```
include "C.Grm"
include "C_ext.Grm"
include "static_slice.txl"
include "pointer.txl"
include "declarations.txl"

function main
      replace [program]
            P [program]

      by
            P [RemovePointersMain][message "Pointers Eliminated."]
[print][breakpoint]
              [static_slice] [message "Static Slice done"]
[print][breakpoint]
              [EliminateDeclarations]
end function
```

main2.txl
```
include "C.Grm"
include "C_ext.Grm"
include "C_overide.Grm"
include "exp.txl"
include "push.txl"
include "kill.txl"
include "declarations.txl"

function main
      replace [program]
            P [program]
      by
            P [PushTrans] [message "Push Trans
done"][print][breakpoint]
              [ResolveExpression][message "ExpressionResolved"]
[print][breakpoint]
              [EliminateKill][message "Eliminate Kill
Stmts"][print][breakpoint]
              [EliminateDeclarations][message
"EliminateDeclaration"][print][breakpoint]
end function
```

static_slicer.txl
```
#if not STATIC_SLICER_TXL
#define STATIC_SLICER_TXL
```

```
include "C.Grm"
include "C_ext.Grm"


function static_slice
        replace [program]
                P [program]
        by
            P [PropagateMarkup]
              [removeUnmarkedStatements]
              [stripMarkup]
              [RegigIfElse] [message "Static Sliced."] [print]
                        % remove empty if body and negate the
                        % condition for else body
end function

rule PropagateMarkupTop
      replace [program]
            P [program]

      construct NP [program]
            P [PropagateMarkup]

      deconstruct not NP
            P
      by
            NP
end rule

rule PropagateMarkup
      replace [program]
            P [program]

      construct NP [program]
            P [backPropagateAssignment]
              [MarkStatementsModifyLoopInvariant] % in while loop
              [whilePropogateMarkup]
%             [whilePropogateMarkupIn]
              [MarkDeepAssignments]
%             [MarkOuterStatement]
%             [MarkOuterStatemetsContainsMarkedStmts]


      % we stop when NP = P.
      deconstruct not NP
            P
      by
            NP
end rule

% Marked statements inside the body of the loop are
% applied on the whole body of the loop for dependency
% markup.
rule whilePropogateMarkup
      replace $ [statement]
            while ( E [expression] ) S [statement]
      construct MarkedS [marked_statement*]
            _ [^ S]
      construct MarkedE [expression*]
            _ [^ MarkedS]
      by
            while ( E )
                  S [markAssignmentsTo each MarkedE]

end rule


rule whilePropogateMarkupIn
      replace $ [statement*]
            while ( E [expression] ) S [statement]
            MoreS [statement*]

      construct MarkedMoreS [marked_statement*]
            _ [^ MoreS]
      construct MarkedMoreE [expression*]
            _ [^ MarkedMoreS]
      by
            while ( E )
                  S [markAssignmentsTo each MarkedMoreE]
            MoreS
end rule

rule IfPropagateMarkupIn
      replace $ [statement*]
            'if ( E [expression] ) { S [statement*]}
            MoreS [statement*]

      construct MarkedMoreS [marked_statement*]
            _ [^ MoreS]
      construct MarkedMoreE [expression*]
            _ [^ MarkedMoreS]
      by
            'if ( E ) {
                  S [markAssignmentsTo each MarkedMoreE]
            }
            MoreS
end rule

rule IfElsePropagateMarkupIn
      replace $ [statement*]
            'if ( E [expression] ) { S [statement*]} 'else {ElseS
[statement*]}
            MoreS [statement*]

      construct MarkedMoreS [marked_statement*]
            _ [^ MoreS]
      construct MarkedMoreE [expression*]
            _ [^ MarkedMoreS]

      by
            'if ( E ) {
                  S [markAssignmentsTo each MarkedMoreE]
            }
            'else {
                  ElseS [markAssignmentsTo each MarkedMoreE]
            }
            MoreS
end rule

% Mark all the statements inside the loop that modify the
% loop invariants.
rule MarkStatementsModifyLoopInvariant
      replace $ [statement]
            Stmt [statement]

      deconstruct Stmt
             'while ( E [expression] ) S [statement]

      where
            S [hasMarkedStmtInside]

      by
            'while ( E )
                  S [markAssignmentsTo E]
end rule

function SubLoopInvariant E [expression]
      replace [statement*]
            All [statement*]

      where
            All [hasMarkedStmtInside]

      by
            All [markAssignmentsTo E]

end function


rule markAssignmentsTo Exp [expression]
      skipping [marked_statement]
      replace [statement*]
            X [id] Op [assignment_operator] E
[assignment_expression];
            More [statement*]

      deconstruct * [id] Exp
            X
      by
            <mark> X Op E ; </mark>
            More
end rule

% Mark any assignment statement deep in a nested loop or if
% and having dependency in the far upperlevel statements
% following the nested.
% eg: if(true){
%     k = 3;
%     }
%     <mark> printf("%d", k)</mark>
% here k= 3 need to be marked.
rule MarkDeepAssignments
      replace $ [statement*]
            S [statement]
            More [statement*]

      construct ExpStmts [expression_statement*]
            _ [^ S]

      construct N [number]
            _ [length ExpStmts]

      deconstruct not N
            0

      by
            S [MarkAssignmentToID More each ExpStmts]
            More
end rule

rule MarkAssignmentToID More [statement*] ExpStmt
[expression_statement]
      skipping [marked_statement]
      replace [statement]
            S [statement]

      deconstruct S
            ExpStmt

      deconstruct S
             X [id] Op [assignment_operator] E
[assignment_expression];

      where
            More [hasMarkedUse X]
                 [hasWhileConditionUse X]
                 [hasIfConditionUse X]
      by
            <mark>S </mark>
end rule


rule backPropagateAssignment
      skipping [marked_statement]

      replace [statement*]
            X [id] Op [assignment_operator] E
[assignment_expression];
            More [statement*]

      where
            More [hasMarkedUse X]
                 [hasWhileConditionUse X]
                 [hasIfConditionUse X]
      by
            <mark> X Op E ; </mark>
            More
```

```
end rule

function hasMarkedUse X [id]
      match * [marked_statement]
            M [marked_statement]

      deconstruct * [expression] M
            E [expression]

      deconstruct * [id] E
            X
end function


function hasIfConditionUse X [id]
      match * [if_statement]
            If [if_statement]

      deconstruct * [marked_statement] If
            _ [marked_statement]

      deconstruct If
            'if ( E [expression] ) _ [statement] _ [opt
else_statement]

      deconstruct * [id] E
            X
end function

function hasWhileConditionUse X [id]
      match * [while_statement]
            W [while_statement]

      deconstruct * [marked_statement] W
            _ [marked_statement]

      deconstruct W
            'while '( E [expression] ') _ [statement]

      deconstruct * [id] E
            X
end function


% Checking a block of statement is having any
% marked statement. this is used when marking
% the statements that modify the loop invariants
% we need to consider them if any part of the loop
% part of the slice.
function hasMarkedStmtInside
      skipping [marked_statement]
      match * [marked_statement]
            _ [marked_statement]

end function

rule removeUnmarkedStatements
```

```
      replace [statement*]
            S [statement]
            More [statement*]
      deconstruct not S
            _ [marked_statement]

      where not
            S [hasMarkedStmtInside]

      by
            More
end rule

rule stripMarkup
      replace [statement]
            < _ [id] > S[statement] </ _ [id] >
      by
            S
end rule

rule MarkOuterStatemetsContainsMarkedStmts
      replace  [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            _ [marked_statement]

      construct Subs [marked_statement*]
            _ [^ S]

      construct N [number]
            _ [length Subs]

      where
            N [ > 0]


      by
            <mark> S </mark>
            More
end rule

rule MarkOuterStatement
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            _ [marked_statement]

      deconstruct * [marked_statement] S
            _ [marked_statement]

      by
            <mark> S </mark>
            More
```

```
end rule

rule RegigIfElse
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            'if ( exp [expression] ) { } 'else { ElseBody [statement]
}

      construct NewIf [statement]
            'if ( ! ( exp ) ) { ElseBody }

      by
            NewIf
            More
end rule

#endif
```

push.txl

```
#if not _PUSH_TXL
#define _PUSH_TXL

include "C.Grm"
include "C_ext.Grm"
include "exp.txl"

% pass in [program] or any other sub set.

function PushTrans
      replace [program]
            P [program]

      construct NP [program]
            P [PushTransform]
              [UnmarkAll]
                [ResolveExpression]
              [ResolveBraces]
              [EliminateUnusedAssignments P]

      deconstruct not P
            NP
      by
            NP
end function

rule PushTransform
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            Id [id] Op [assignment_operator] AssExp
```

```
[conditional_expression];

      construct UnaryE [unary_expression]
            ( AssExp )

      construct ExpIds [id*]
            _ [^ AssExp]

      construct DepIDs [id*]
            ExpIds [. Id]

      construct New [statement*]
            '<pushed> S '</pushed>
            More [MarkForReassign DepIDs]
                 [MarkStatementsAfterReassign]
                 [MarkStatementsFollowingReassignInSide]
                 [MarkStatementsInWhileLoop]
                 [ApplyOnUnmarkedStatements Id UnaryE]
                 [ApplyOnUnmarkedWhile Id UnaryE]
                 [ApplyOnUnmarkedIf Id UnaryE]
%                [ApplyOnUnmarkedIfElse Id UnaryE]
                 [UnmarkAll]

      where not
            AssExp [hasProgramUse Id]

      deconstruct not New
            S More
      by
            New
end rule

rule MarkForReassign DepIDs [id*]
      skipping [marked_statement]
      replace [statement]
            S [statement]
      deconstruct S
            Id [id] Op [assignment_operator] AssExp
[conditional_expression];

      where
            Id [Equals each DepIDs]
      by
            '<reassign> S '</reassign>
end rule

% Check for equality between two IDs. one pass in as parameter,
% Other as the the tree to apply.
function Equals Id [id]
      match * [id]
            Id
end function


% Mark statements that does not suite for push transformation.
% -----------------------------------------------------------
% In a block of statement, if one statement is marked as reassigned
% then mark all the rest to avoid any push transformation on them.
rule MarkStatementsAfterReassign
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            '<reassign> _ [statement] '</reassign>

      construct New [statement*]
            S
            More [MarkAll]
                 [MarkWhileStatements]
                [MarkIfStatements]

      deconstruct not New
            S More
      by
            New
end rule

rule MarkStatementsFollowingReassignInSide
      replace [statement*]
            S [statement]
            More [statement*]

      where
            S [HasReassignStatementInside]

      construct New [statement*]
            S
            More [MarkAll]
                 [MarkWhileStatements]
                [MarkIfStatements]

      deconstruct not New
            S More
      by
            New
end rule

%IF the body of a loop has reassign, then mark all the statements in
% the body. so none of them will be get pushed.
rule MarkStatementsInWhileLoop
      replace $ [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            'while '( Expression [expression] ') Body [statement]

      where
            Body [HasReassignStatementInside]

      construct NewS [statement]
            S [MarkAll]
```

```
                 [MarkWhileStatements]
                [MarkIfStatements]

      by
            '<push_mark> NewS '</push_mark>
            More
end rule

% Check for any reassign statement inside the passed in block of
% statements.
function HasReassignStatementInside
      match * [marked_statement]
            S [marked_statement]

      deconstruct S
            '<reassign> _ [statement] '</reassign>
end function

rule MarkAllExceptIfWhile
      replace $ [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            '<push_mark> _ [statement] '</push_mark>
      deconstruct not S
            _ [if_statement]
      deconstruct not S
            _ [while_statement]

      by
            '<push_mark> S '</push_mark>
            More
end rule

rule MarkAll
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            _ [marked_statement]
      by
            '<push_mark> S '</push_mark>
            More
end rule

rule MarkWhileStatements
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            _ [while_statement]
      by
            '<push_mark> S '</push_mark>
            More
```

```
end rule

rule MarkIfStatements
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            _ [if_statement]
      by
            '<push_mark> S '</push_mark>
            More
end rule


% ----------------------------------------------------------------

rule UnmarkAll
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            '< Tag [id] '> InnerS [statement]
                        '</ Tag '>
      by
            InnerS
            More
end rule

rule ApplyOnUnmarkedStatements Id [id] ReplaceUnaryE
[unary_expression]
      replace  [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            _ [marked_statement]

      deconstruct not S
            _ [while_statement]


      deconstruct not S
            _ [if_statement]

      construct New [statement]
            S [ReplaceUnaryExpression Id
ReplaceUnaryE]

      deconstruct not New
            S
      by
            '<pushedin> New '</pushedin>
            More
end rule
```

```
rule ApplyOnUnmarkedWhile Id [id] ReplaceUnaryE [unary_expression]
      replace  [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            _ [marked_statement]

      deconstruct S
            'while ( exp [expression] ) Body [statement]

      construct New [expression]
            exp [ReplaceUnaryExpression Id ReplaceUnaryE]

      deconstruct not New
            exp
      by
            'while ( New ) Body
            More
end rule

rule ApplyOnUnmarkedIf Id [id] ReplaceUnaryE [unary_expression]
      replace  [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            _ [marked_statement]

      deconstruct S
            'if ( exp [expression] ) IfBody [statement] ElseBody [opt
else_statement]

      construct New [expression]
            exp [ReplaceUnaryExpression Id ReplaceUnaryE]

      deconstruct not New
            exp
      by
            'if ( New ) IfBody ElseBody
            More
end rule

rule ApplyOnUnmarkedIfElse Id [id] ReplaceUnaryE [unary_expression]
      replace  [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            _ [marked_statement]

      deconstruct S
            'if ( exp [expression] ) IfBody [statement] 'else
ElseBody [statement]

      construct New [expression]
            exp [ReplaceUnaryExpression Id ReplaceUnaryE]
      deconstruct not New
            exp
      by
            'if ( New ) IfBody 'else ElseBody
            More
end rule


rule OLD__ApplyOnUnmarkedStatements Id [id] ReplaceUnaryE
[unary_expression]
      skipping [marked_statement]
      replace  [statement]
            S [statement]

      deconstruct not S
            _ [marked_statement]

      construct New [statement]
            S [ReplaceUnaryExpression Id ReplaceUnaryE]

      deconstruct not New
            S
      by
            New
end rule

rule ResolveBraces
      replace [unary_expression]
            '( IdOrNum [reference] ')
      by
            IdOrNum
end rule


function ReplaceUnaryExpression I [id] E [unary_expression]
    replace * [conditional_expression]
        P [conditional_expression]
    by
        P [mark I]
          [sweep E]
end function

rule mark I [id]
    replace [postfix_expression]
        I
    construct M [marked_reference]
        I
    by
        M
end rule

rule sweep E [unary_expression]
    replace [postfix_expression]
        M [marked_reference]

    by
```

```
        ( E )
end rule




rule EliminateUnusedAssignments P [program]
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            Id [id] Op [assignment_operator] AssExp
[conditional_expression];

      where not
            P [hasProgramUse Id]
      by
            More
end rule

function hasProgramUse X [id]
      match * [conditional_expression]
            E [conditional_expression]

      deconstruct * [id] E
            X
end function

#endif
```

kill.txl

```
#if not _KILL_TXL
#define _KILL_TXL

include "C.Grm"
include "C_ext.Grm"
include "C_overide.Grm"
include "exp.txl"

% pass in [program] or any other sub set.

function EliminateKill
      replace [program]
            P [program]

      by
            P [SimpleKillEliminate]
              [KILL_MarkAssignmentsUsedIn]
              [KILL_MarkAssignmentsUsedOutSide]
              [KILL_MarkAssignmetsUsedInLoop]
              [KILL_DeleteNonMarkedAssignments]
              [KILL_UnmarkAll]
end function

% ==============================================================
% Eliminate a Kill statement in a single sequence of statements.
% Apply this first, to eliminate errors at the presence of loops.
% bcoz, extrating all assignments and mark for being used outside
% will left any killed statement also in the slice.
rule SimpleKillEliminate
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            _ [marked_statement]
      deconstruct S
            Id [id] Op [assignment_operator] AssExp
[conditional_expression];

      % Mark the reassignment statement first, and mark all
      % the statements following that. This is done in a copy
      % not in the original. so no need to unmark.
      construct StmtsBeforeReassign [statement*]
            More [KILL_MarkForReassign Id]
                [KILL_MarkStatementsAfterReassign]
                [KILL_DeleteMarked] % delete only <kill_mark>'s

      % Check whether Id is used in the current block
      % between this assignment and any reassignment.
      % by checking all expression in the unmarked stmts.
      where not
            StmtsBeforeReassign [KILL_IdUsedInStatements Id]

      % Check is there any reassignment really, and not just
      % an unused assignment. Do not allow an unused assignment
      % at this time. as it may be used in an outer block.
      where
            StmtsBeforeReassign [KILL_IsReassignStmt each
StmtsBeforeReassign]

      by
            More
end rule

% Check the provided statement is marked as a reassign.
function KILL_IsReassignStmt Stmt [statement]
      % Match part is not useful here. only the deconstruct part is
      % doing the test.
      match [statement*]
            _ [statement*]

      deconstruct Stmt
            '<reassign> InnerS [statement] '</reassign>
end function

%Check the given Id is used in the Passed statement*
function KILL_IdUsedInStatements Id [id]
      match * [conditional_expression]
            E [conditional_expression]

      deconstruct * [id] E
```

```
            Id
end function

%     END of simple kill statement elimination
%=================================================================


% Mark Assignments inside a loop, which is used any other
% expression inside the loop or in the loop condition.
% So it will not be eliminated.
rule KILL_MarkAssignmetsUsedInLoop
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            _ [while_statement]

      % extract all the assignment statements and if it is used
      % inside the loop anywhere or in the conditional exp then
      % mark it as <loop>
      construct Assmts [statement*]
            _ [ ^ S]

      % All the expressions in the loop, including the looping
      % condition expression
      construct Exps [conditional_expression*]
            _ [ ^ S]

      % New loop statement and following statements appended,
      % inside loop statements marked, if it required.
      construct New [statement*]
            S [KILL_MarkAssignmentIfUsedInLoop Exps each Assmts]
            More

      deconstruct not New
            S More
      by
            New
end rule

% Exp holds all the expressions inside the loop and the loop
condition.
% This function test 'Assmt' is used in 'Exp', if so mark 'Assmt'
inside
% the loop.
function KILL_MarkAssignmentIfUsedInLoop Exp
[conditional_expression*] Assmt [statement]
      replace [statement]
            S [statement]

      deconstruct not Assmt
            _ [marked_statement]
      deconstruct Assmt
            Id [id] Op [assignment_operator] AssExp
[conditional_expression];
```

```
      deconstruct * [id] Exp
            Id
% Hotpoint: above two lines replace the below.
%     where
%           Exp [KILL_hasProgramUse Id]
      by
            S [KILL_MarkThisStatementInLoop Assmt]
end function

rule KILL_MarkThisStatementInLoop AssStmt [statement]
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            AssStmt
      by
            '<loop> S '</loop>
            More
end rule
% End of Loop marking
%---------------------------------------------------------------


% Mark statements those are used down, before any reassignments.
% Handles...
%  var = 3;
%  if(...){
%     var = 5;
%     printf("%d", var);
%  }
% //var not used here after.
% var in the top is not marked as used.
% if var is used below if, then var is marked as required.
rule KILL_MarkAssignmentsUsedIn
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            Id [id] Op [assignment_operator] AssExp
[conditional_expression];

      construct Marked [statement*]
            More [KILL_UnmarkAll][KILL_MarkForReassign Id]
                [KILL_MarkStatementsAfterReassign]
                [KILL_DeleteMarked]% delete all marked as <kill_mark>

      where
            Marked [KILL_IsARequiredStmt Id]

      by
            '<internal> S '</internal>
            More
end rule
```

```
% Handles an Internal assignment not being used outside below. and
% also inside below.
% Handles...
% if(...){
%     var = 30;
% }
% printf("%d", var);
% // var inside will be marked as required. if var is not used
outside
% // then var will not be marked. left to be deleted.
rule KILL_MarkAssignmentsUsedOutSide
      replace [statement*]
            All [statement*]

      deconstruct All
            S [statement]
            More [statement*]

      % Extract all assignments in a block statement S (if it is)
      construct AssStmts [statement*]
            _ [^ S]

      construct New [statement*]
            All [KILL_MarkAssignmentsUsedOutSideSub each AssStmts]

      deconstruct not All
            New
      by
            New
end rule

function KILL_MarkAssignmentsUsedOutSideSub AssStmt [statement]
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct AssStmt
            Id [id] Op [assignment_operator] AssExp
[conditional_expression];

      construct Marked [statement*]
            More [KILL_UnmarkAll][KILL_MarkForReassign Id]
                [KILL_MarkStatementsAfterReassign]
%               [KILL_MarkStatementsFollowingReassignInSide]
                [KILL_DeleteMarked] % delete all marked as
<kill_mark>

      where
            Marked [KILL_IsARequiredStmt Id]

      by
            S [KILL_MarkThisStatement AssStmt]
            More
end function


rule KILL_MarkThisStatement AssStmt [statement]
```

```
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            AssStmt
      by
            '<ext> S '</ext>
            More
end rule


function KILL_IsARequiredStmt Id [id]
      match * [conditional_expression]
            E [conditional_expression]

      deconstruct * [id] E
            Id
end function


% General Functions used in all 3 above major functions.
% Marking statements following reassign, mark for reassign etc.
%---------------------------------------------------------------
rule KILL_DeleteMarked
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            '<kill_mark> _ [statement] '</kill_mark>
      by
            More
end rule

rule KILL_MarkForReassign Id [id]
      skipping [marked_statement]
      replace [statement]
            S [statement]
      deconstruct S
            Id Op [assignment_operator] AssExp
[conditional_expression];

      by
            '<reassign> S '</reassign>
end rule

% Mark statements that does not suite for push transformation.
% ---------------------------------------------------------

% In a block of statement, if one statement is marked as reassigned
% then mark all the rest to avoid any push transformation on them.
rule KILL_MarkStatementsAfterReassign
      replace [statement*]
            S [statement]
            More [statement*]
      deconstruct S
            '<reassign> _ [statement] '</reassign>

      construct New [statement*]
            S
            More [KILL_MarkAll]
                 [KILL_MarkWhileStatements]
                [KILL_MarkIfStatements]

      deconstruct not New
            S More
      by
            New
end rule

% if a block statement has a reassign inside, mark all the statements
% that follows the block stmt.
% NOT applicable to KILL statement elimination.
% NOT USED here.
rule KILL_MarkStatementsFollowingReassignInSide
      replace [statement*]
            S [statement]
            More [statement*]

      where
            S [KILL_HasReassignStatementInside]

      construct New [statement*]
            S
            More [KILL_MarkAll]
                 [KILL_MarkWhileStatements]
                [KILL_MarkIfStatements]

      deconstruct not New
            S More
      by
            New
end rule

% Check for any reassign statement inside the passed in block of
% statements.
function KILL_HasReassignStatementInside
      match * [marked_statement]
            S [marked_statement]

      deconstruct S
            '<reassign> _ [statement] '</reassign>
end function

rule KILL_MarkAll
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            _ [marked_statement]
      by
```

```
            '<kill_mark> S '</kill_mark>
            More
end rule

rule KILL_MarkWhileStatements
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            _ [while_statement]
      by
            '<kill_mark> S '</kill_mark>
            More
end rule

rule KILL_MarkIfStatements
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            _ [if_statement]
      by
            '<kill_mark> S '</kill_mark>
            More
end rule
% -----------------------------------------------------------------

rule KILL_UnmarkAll
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            '< Tag [id] '> InnerS [statement]
                        '</ Tag '>
      by
            InnerS
            More
end rule


% Delete all the assignments those are not marked as relevant to the
slice.
% kill statement in the same sequence is marked as <internal> and
kill
% statement in an outter level, and not used between are
rule KILL_DeleteNonMarkedAssignments
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            _ [marked_statement]

      deconstruct S
```

```
            Id [id] Op [assignment_operator] AssExp
[conditional_expression];

      by
            More
end rule


% NOT being used.
function KILL_hasProgramUse X [id]
      match * [conditional_expression]
            E [conditional_expression]

      deconstruct * [id] E
            X
end function

#endif
```

exp.txl

```
#if not _EXP_TXL
#define _EXP_TXL

include "C.Grm"
include "C_overide.Grm"

% pass in [program] or any other sub set.

%function main
%     replace [program]
%           P [program]
%     by
%           P [ResolveExpression]
%end function

function ResolveExpression
      replace [program]
            P [program]
      by
            P  [EliminateCastAdd]
               [EliminateCastAdd2]
               [EliminateCastMul]
               [EliminateCastMul2]
               [ResolveMul]
               [ResolveMul2]
               [CoefficientFirst]
               [ResolveAdd]
               [ResolveAdd2]
               [AddZeroToAdditive]
               [AddUnitCoefficient]
               [AddTogetherEquals]
               [EliminateZeros]
               [EliminateUnitCoefficient]
               [EXP__ResolveBraces]
end function
```

```
%Eliminate any unnecessary cast around the first element
% of an addtive expression
rule EliminateCastAdd
      replace [additive_expression]
            Mul [multiplicative_expression]
            AddSubMuls [add_subtract_multiplicative_expression*]

      deconstruct Mul
            ( AdditiveE [additive_expression] )

%     construct dummy2 [additive_expression]
%           AdditiveE [debug]

      deconstruct AdditiveE
            innerMul [multiplicative_expression]
            innerAddSubMuls [add_subtract_multiplicative_expression*]

      construct JoinedASM [add_subtract_multiplicative_expression*]
            innerAddSubMuls [. AddSubMuls]

      by
            innerMul JoinedASM
end rule

%Eliminate any unnecessary cast around the following elements
% of an additive expression
rule EliminateCastAdd2
      replace [add_subtract_multiplicative_expression*]
            ASMone [add_subtract_multiplicative_expression]
            More [add_subtract_multiplicative_expression*]

      deconstruct ASMone
            AddOp [additive_operator] Mul [multiplicative_expression]

      deconstruct Mul
             ( AdditiveE [additive_expression] )

      deconstruct AdditiveE
            innerMul [multiplicative_expression]
            innerAddSubMuls [add_subtract_multiplicative_expression*]

      construct FirstInnerASM
[add_subtract_multiplicative_expression]
            AddOp innerMul

      construct JoinedASM [add_subtract_multiplicative_expression*]
            FirstInnerASM  innerAddSubMuls

      by
            JoinedASM [. More]
end rule

%Eliminate the cast around the first unary of the
% multiplicative expression.
rule EliminateCastMul
      replace [multiplicative_expression]
```

```
            Mul [multiplicative_expression]


%     construct dummy [multiplicative_expression]
%           Mul [debug]

      deconstruct Mul
            ( UnaryE [unary_expression]
              MulDivCast [multipy_divide_cast_expression*] )
            MulDivCastExt [multipy_divide_cast_expression*]

      by
            UnaryE MulDivCast [. MulDivCastExt]
end rule

% Eliminate the cast around any following multiplicative expression.
rule EliminateCastMul2
      replace [multipy_divide_cast_expression*]
            MDCOne [multipy_divide_cast_expression]
            More [multipy_divide_cast_expression*]

      deconstruct MDCOne
            MulOp [multiplicative_operator]
            ( Mul [multiplicative_expression] )

      deconstruct Mul
            UnaryE [unary_expression]
            InnerMDC [multipy_divide_cast_expression*]

      construct MDCNew [multipy_divide_cast_expression]
            MulOp UnaryE

      by
            MDCNew InnerMDC [. More]

end rule


%Resolve Numericals
%------------------

%Resolve Multiplication
rule ResolveMul
      replace $ [multiplicative_expression]
            Mul [multiplicative_expression]

      deconstruct Mul
            UnaryE [unary_expression]
            MDCs [multipy_divide_cast_expression*]

      deconstruct UnaryE
            UnaryVal [number]

      by
            UnaryVal [GetMultiplied each MDCs]
            MDCs [EliminateMulTerms]
end rule
```

```
rule ResolveMul2
      replace $ [multiplicative_expression]
            Mul [multiplicative_expression]

      deconstruct Mul
            UnaryE [unary_expression]
            MDCs [multipy_divide_cast_expression*]

      deconstruct not UnaryE
            _ [number]

      by
            UnaryE MDCs[GetMultiplied2]
end rule


% Multiply the first Unary of the multiplicative expression by
% each mul_div_cast_exp in the list, which is a number.
% the final val is replaced at the first Unary.
function GetMultiplied aMDC [multipy_divide_cast_expression]
      replace [number]
            N [number]

%     construct dummy [multipy_divide_cast_expression]
%           aMDC [debug]

      deconstruct aMDC
            Op [multiplicative_operator]
            N2 [number]
      by
            N [Mul Op N2]
              [Div Op N2] % only of this will be effective. depend on
the
                        % operator is * or /
end function

% find the first number in the multiplicative_exp and mul it
% by other following numbers. and eliminate them.
rule GetMultiplied2
      replace  $ [multipy_divide_cast_expression*]
            First [multipy_divide_cast_expression]
            More [multipy_divide_cast_expression*]

      deconstruct First
            Op [multiplicative_operator] N [number]

      construct New [multipy_divide_cast_expression]
            Op N [GetMultiplied each More]

      by
            New More[EliminateMulTerms]
end rule


% If the provided operator is a Mul, then it multiply
function Mul Op [multiplicative_operator] N2 [number]
```

```
      replace [number]
            N [number]

      deconstruct Op
            '*
      by
            N [* N2]
end function

% If the provided operator is a Div then divide
function Div Op [multiplicative_operator] N2 [number]
      replace [number]
            N [number]

      deconstruct Op
            '/
      by
            N [/ N2]
end function

% Once all the nums in the mul_div_cast_exp are resolved
% and replaced at the first element, remove them all.
rule EliminateMulTerms
      replace [multipy_divide_cast_expression*]
            First [multipy_divide_cast_expression]
            More [multipy_divide_cast_expression*]

      deconstruct First
            Op [multiplicative_operator]
            N [number]
      by
            More
end rule

%Bring the numeric factor to the first
rule CoefficientFirst
      replace $ [multiplicative_expression]
            UnaryE [unary_expression]
            MDCs [multipy_divide_cast_expression*]

      construct Unarytmp [unary_expression]
            UnaryE      % a copy of unaryE
      by
            Unarytmp [CoefficientFirstSub2 each MDCs]
            MDCs[CoefficientFirstSub UnaryE]


end rule

rule CoefficientFirstSub UnaryE [unary_expression]
      replace [multipy_divide_cast_expression*]
            First [multipy_divide_cast_expression]
            More [multipy_divide_cast_expression*]

      deconstruct First
            Op [multiplicative_operator]
            N [number]
      by
            Op UnaryE
            More
end rule

function CoefficientFirstSub2 aMDC [multipy_divide_cast_expression]
      replace [unary_expression]
            UnaryE [unary_expression]

      deconstruct aMDC
            Op [multiplicative_operator] N [number]
      by
            N
end function


% Resolve Additions
%-------------------
rule ResolveAdd
      replace $ [additive_expression]
            AddE [additive_expression]

      deconstruct AddE
            MulE [multiplicative_expression]
            ASMs [add_subtract_multiplicative_expression*]


      deconstruct MulE
            MulEVal [number]

      by
            MulEVal [GetAdded each ASMs]
            ASMs [EliminateAddTerms]
end rule

rule ResolveAdd2
      replace $ [additive_expression]
            AddE [additive_expression]

      deconstruct AddE
            MulE [multiplicative_expression]
            ASMs [add_subtract_multiplicative_expression*]

      deconstruct not MulE
            _ [number]

      by
            MulE ASMs [GetAdded2]
end rule


% Multiply the first Unary of the multiplicative expression by
% each mul_div_cast_exp in the list, which is a number.
% the final val is replaced at the first Unary.
function GetAdded aASM [add_subtract_multiplicative_expression]
      replace [number]
```

```
            N [number]

      deconstruct aASM
            Op [additive_operator]
            N2 [number]
      by
            N [Add Op N2]
              [Sub Op N2] % only of this will be effective. depend on
the
                        % operator is * or /
end function

% find the first number in the multiplicative_exp and mul it
% by other following numbers. and eliminate them.
rule GetAdded2
      replace $ [add_subtract_multiplicative_expression*]
            First [add_subtract_multiplicative_expression]
            More [add_subtract_multiplicative_expression*]

      deconstruct First
            Op [additive_operator] N [number]

      construct New [add_subtract_multiplicative_expression]
            Op N [GetAdded each More]

      by
            New More[EliminateAddTerms]
end rule

% If the provided operator is a Mul, then it multiply
function Add Op [additive_operator] N2 [number]
      replace [number]
            N [number]

      deconstruct Op
            '+
      by
            N [+ N2]
end function

% If the provided operator is a Div then divide
function Sub Op [additive_operator] N2 [number]
      replace [number]
            N [number]

      deconstruct Op
            '-
      by
            N [- N2]
end function

% After adding all the num terms in the ASM_exp list,
% eliminate them from the list here.
rule EliminateAddTerms
      replace [add_subtract_multiplicative_expression*]
            First [add_subtract_multiplicative_expression]
            More [add_subtract_multiplicative_expression*]
      deconstruct First
            Op [additive_operator]
            N [number]
      by
            More
end rule


%Resolve Similar Expressions
%----------------------------
rule AddTogetherEquals
      replace $ [additive_expression]
            Mul [multiplicative_expression]
            ASMs [add_subtract_multiplicative_expression*]

      by
            Mul ASMs[AddTogetherEqualsSub]
end rule

rule AddTogetherEqualsSub
      replace $ [add_subtract_multiplicative_expression*]
            First [add_subtract_multiplicative_expression]
            More [add_subtract_multiplicative_expression*]

      deconstruct First
            Op [additive_operator] MulE [multiplicative_expression]

      by
            Op MulE [GetExpressionMultiplied each More]
            More   [EliminateExpressionTerms MulE]
end rule

function GetExpressionMultiplied aASM
[add_subtract_multiplicative_expression]
      replace $ [multiplicative_expression]
            MulOne [multiplicative_expression]

      deconstruct MulOne
            N [number] MDCs [multipy_divide_cast_expression*]

      deconstruct aASM
            Op [additive_operator]  MulTwo
[multiplicative_expression]

      deconstruct MulTwo
            N2 [number] MDCs2 [multipy_divide_cast_expression*]

      where MulOne [CheckEqual MulTwo]

      by
            N [Add Op N2] MDCs
end function

rule EliminateExpressionTerms MulOne [multiplicative_expression]
      replace [add_subtract_multiplicative_expression*]
            First [add_subtract_multiplicative_expression]
            More [add_subtract_multiplicative_expression*]

      deconstruct First
            Op [additive_operator]
            MulTwo [multiplicative_expression]

      where
            MulOne [CheckEqual MulTwo]
      by
            More
end rule


rule AddZeroToAdditive
      replace [additive_expression]
            Mul [multiplicative_expression]
            ASMs [add_subtract_multiplicative_expression*]

      deconstruct not Mul
            N [number]

      construct Zero [multiplicative_expression]
            0
      by
            Zero '+ Mul ASMs
end rule

rule AddUnitCoefficient
      replace [multiplicative_expression]
            UnaryE [unary_expression]
            MDCs [multipy_divide_cast_expression*]

      deconstruct not UnaryE
            N [number]

      construct UnitE [unary_expression]
            1

      construct NewMul [multiplicative_expression]
            UnitE '* UnaryE MDCs


      by
            NewMul

end rule

function CheckEqual MulExpRight [multiplicative_expression]
      match [multiplicative_expression]
            UnaryE [unary_expression]
            MDCs [multipy_divide_cast_expression*]

      deconstruct MulExpRight
            UnaryERight [unary_expression]
            MDCsRight [multipy_divide_cast_expression*]

      construct N1 [number]
```

```
            _ [length MDCs]

      construct N2 [number]
            _ [length MDCsRight]

      deconstruct N1
            N2

      where all
            MDCs [ ContainsMDC each  MDCsRight]
end function

function ContainsMDC MDCRight [multipy_divide_cast_expression]
      skipping [multipy_divide_cast_expression]
      match * [multipy_divide_cast_expression]
            MDCRight
end function


rule EliminateZeros
      replace [additive_expression]
            Mul [multiplicative_expression]
            ASMs [add_subtract_multiplicative_expression*]

      deconstruct ASMs
            First [add_subtract_multiplicative_expression]
            More [add_subtract_multiplicative_expression*]

      deconstruct First
            Op [additive_operator] M [multiplicative_expression]

      deconstruct Op
            '+

      deconstruct Mul
            N [number]
      by
            M More
end rule

rule EliminateUnitCoefficient
      replace [multiplicative_expression]
            UnaryE [unary_expression]
            MDCs [multipy_divide_cast_expression*]

      deconstruct UnaryE
            1

      deconstruct MDCs
            First [multipy_divide_cast_expression]
            More [multipy_divide_cast_expression*]

      deconstruct First
            Op [multiplicative_operator] U [unary_expression]

      by
            U More
```

```
end rule

rule EXP__ResolveBraces
      replace [unary_expression]
            '( IdOrNum [reference] ')
      by
            IdOrNum
end rule


#endif
```

declarations.txl

```
#if not DECLARATIONS_TXL
#define DECLARATIONS_TXL

include "C.Grm"

%rule EliminateDeclarations
rule EliminateDeclarations
      replace [program]
            P [program]

      construct NP [program]
            P [EliminateUnusedVarDeclarations]
              [EliminateEmptyDeclaration]

      deconstruct not NP
            P
      by
            NP
end rule


rule EliminateUnusedVarDeclarations
      replace [compound_statement_body]
            Decl [declaration]
            Body [compound_statement_body]

      deconstruct Decl
            Spec [decl_specifiers] DeclList [list init_declarator];

      construct NewList [list init_declarator]
            DeclList[EliminateRedundants Body]

      deconstruct not DeclList
            NewList

      by
            Spec NewList;
            Body
end rule

rule EliminateEmptyDeclaration
      replace $ [compound_statement_body]
            Decl [declaration]
            Body [compound_statement_body]

      deconstruct Decl
            Spec [decl_specifiers] DeclList [list init_declarator];

      construct N [number]
            _ [length DeclList]

      deconstruct N
            0

      by
            Body
end rule

%Examine each variable on a multiple var declaration
% and eliminate if any of them are not being used, from the list.
rule EliminateRedundants Body [compound_statement_body]
      replace [list init_declarator]
            First [init_declarator],
            More [list init_declarator]

      deconstruct First
            _ [ptr_operator*] Id [id]


      where not
            Body [IsUsingVariable Id]

      by
            More
end rule

% Checking weather the variable Id is being used in the body of the
% declaration. returns true if it is.
rule IsUsingVariable Id [id]
      match * [expression]
            E [expression]

      deconstruct * [id] E
            Id
end rule


% Eliminate Var declaration where only one var in one declaration.
% NOT BEING USED...
%handled in the general case.
rule EliminateSub2
      replace [compound_statement_body]
            Decl [declaration]
            Body [compound_statement_body]
      deconstruct Decl
            Spec [decl_specifiers] DecList[list init_declarator];

      deconstruct DecList
            Id [id]
      where not
            Body [IsUsingVariable Id]

      by
            Body
end rule

#endif
```

pointers.txl

```
#if not POINTER_TXL
#define POINTER_TXL

include "C.Grm"
include "C_ext.Grm"
include "if_else.txl"

function RemovePointersMain
      replace [program]
            P [program]
      by
            P [RemovePointers]
              [EliminateAddressAssignments]
              [RemoveProcessedTag]
              [RemoveCopyTag]
              [RemoveEmptyElse]
              [RemoveEmptyIfElse]
end function

rule RemovePointers
      replace [program]
            P [program]

      construct New [program]
            P [ReplaceDereference]
              [ReplaceWithIfElse]
              [UnmarkAddressAssignments]

      deconstruct not New
            P
      by
            New
end rule

rule ReplaceWithIfElse
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            IfStmt [if_statement]

       deconstruct IfStmt
                 'if ( exp [expression] ) IfBody [statement] _else
[opt
else_statement]
```

```
        construct NewMore [statement*]
                More [ReplaceWithIf exp IfBody]
                     [ReplaceWithElse IfStmt]
                     [ReplaceWithNegatedIf IfStmt]
                     [EliminateEmptyElse]

        deconstruct not NewMore
                More

      where not
            More [HasAddressAssign]

        by
                S [MarkAddressAssignments]
                NewMore
end rule

rule ReplaceDereference
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
             Pointer [unary_expression] _op [assignment_operator] '&
Target [unary_expression];

      where not
            More [HasAddressAssign]
      by
            '<done> S '</done>
            More [Replace Pointer Target]
                 [ReplaceInMarked Pointer Target]
end rule

function HasAddressAssign
      skipping [marked_statement]

      match * [statement]
            S [statement]

      deconstruct S
      _ [unary_expression] _ [assignment_operator] '& t
[unary_expression];

end function

rule Replace Pointer [unary_expression] Target [unary_expression]
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct not S
            '<done> InnerS [statement] '</done>

      construct New [statement]
            S [ReplaceSub Pointer Target]
      deconstruct not S
            New
      by
            New
            More
end rule

rule ReplaceInMarked Pointer [unary_expression] Target
[unary_expression]
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            '<done> InnerS [statement] '</done>

      construct New [statement]
            InnerS [ReplaceSub Pointer Target]

      deconstruct not New
            InnerS

      by
            New
            More
end rule



rule ReplaceSub Pointer [unary_expression] Target [unary_expression]
      replace [unary_expression]
            U [unary_expression]

      deconstruct U
            UnaryOp [unary_operator] Pointer

      deconstruct UnaryOp
            '*
      by
            Target
end rule

rule UnmarkAddressAssignments
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            '<done> InnerS [statement] '</done>
      by
            InnerS
            More
end rule

rule EliminateAddressAssignments
      replace [statement*]
```

```
            S [statement]
            More [statement*]

      deconstruct S
      _ [unary_expression] _ [assignment_operator] '& t
[unary_expression];

      by
            More
end rule

rule RemoveEmptyElse
        replace [statement*]
                stmt [if_statement]
                Rest [statement*]

        deconstruct stmt
                'if ( exp [expression] ) if_body [statement] _else
[opt
else_statement]

        deconstruct _else
                'else { }

        by
                'if ( exp ) if_body
                Rest
end rule

rule RemoveEmptyIfElse
        replace [statement+]
                stmt [if_statement]
                Rest [statement*]

        deconstruct stmt
                'if( exp [expression] ) {}
        by
                Rest
end rule

#endif
```

if_else.txl

```
#if not IF_ELSE_TXL
#define IF_ELSE_TXL

include "C.Grm"
include "C_ext.Grm"

define add_assign_statement
       [unary_expression] [assignment_operator] '&
[unary_expression];
end define

redefine statement
```

```
      [add_assign_statement]
      | ...
end redefine

function ReplaceConditionalDereferenceMain
      replace [program]
            p [program]
      by
            p [ReplaceConditionalDereference]
end function

rule ReplaceConditionalDereference
      replace [statement*]
            IfStmt [if_statement]
            More [statement*]

      deconstruct IfStmt
             'if ( exp [expression] ) IfBody [statement] _else [opt
else_statement]

      construct NewMore [statement*]
            More [ReplaceWithIf exp IfBody]
                 [ReplaceWithElse IfStmt]
                 [ReplaceWithNegatedIf IfStmt]
                 [EliminateEmptyElse]

      deconstruct not NewMore
            More
      by
            IfStmt [MarkAddressAssignments]
            NewMore
end rule


rule ReplaceWithIf exp [expression] IfBody [statement]
      replace [statement*]
            Stmt [statement]
            More [statement*]

      construct AddAssignStmts [add_assign_statement*]
            _ [^ IfBody]

      construct NewStmt [statement]
            Stmt [ReplaceDereferenceIf each AddAssignStmts]


      deconstruct not Stmt
            NewStmt

      construct NewIf [if_statement]
            'if (exp) { NewStmt } 'else { '<copy> Stmt '</copy> }

      by
            '<processed> NewIf '</processed>
            More
end rule
```

```
rule ReplaceWithElse IfStmt [if_statement]
      deconstruct IfStmt
             'if ( exp [expression] ) IfBody [statement] Else [opt
else_statement]

      deconstruct Else
            'else ElseBody [statement]

      construct AddAssignStmts [add_assign_statement*]
            _ [^ ElseBody]

      replace [else_statement]
            'else { MarkedS [marked_statement] }

      deconstruct MarkedS
            '<copy> InnerS [statement] '</copy>

      construct NewS [statement]
            InnerS [ReplaceDereferenceIf each AddAssignStmts]

      deconstruct not NewS
            InnerS
      by
            'else { NewS }
end rule

rule ReplaceWithNegatedIf IfStmt [if_statement]
      deconstruct IfStmt
             'if ( exp [expression] ) IfBody [statement] _else [opt
else_statement]

      deconstruct _else
            'else ElseBody [statement]

      construct AddAssignStmts [add_assign_statement*]
            _ [^ ElseBody]

      replace [statement*]
            _stmt [statement]
            More [statement*]

      construct stmt_n [statement]
            _stmt [ReplaceDereferenceIf each AddAssignStmts]

      where not
            stmt_n [=_stmt]
      by
            'if ( !(exp) ) { stmt_n}
            More
end rule

rule ReplaceDereferenceIf _add_assign_stmt [add_assign_statement]
      skipping [marked_statement]
      replace [unary_expression]
            '* p [unary_expression]

      deconstruct _add_assign_stmt
```

```
             p _ [assignment_operator] '& _t [unary_expression];

      by
            _t
end rule

rule EliminateEmptyElse
      replace [statement*]
            IfStmt [if_statement]
            More [statement*]

      deconstruct IfStmt
             'if ( exp [expression] ) IfBody [statement] _else [opt
else_statement]

%     deconstruct _else
%           'else ElseBody [marked_statement]

      deconstruct _else
            'else '<copy> InnerS [statement] '</copy>

      construct NewIf [if_statement]
            'if ( exp ) IfBody 'else InnerS

      by
            NewIf
            More
end rule

rule MarkAddressAssignments
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            _ [add_assign_statement]

      by
            '<done> S '</done>
end rule

rule RemoveProcessedTag
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            '<processed> InnerS [statement] '</processed>

      by
            InnerS
            More
end rule

rule RemoveCopyTag
      replace [statement*]
            S [statement]
            More [statement*]

      deconstruct S
            '<copy> InnerS [statement] '</copy>

      by
            InnerS
            More
end rule

#endif
```