Modelling
Dynamic Memory Allocation
and
Deallocation using
Amorphous Slicing

by

Yoga Sivagurunathan
BSc. (University of North London)
A thesis submitted in partial fulfilment of the
requirements of the London Metropolitan University
for the degree of

Doctor of Philosophy

January 23, 2004
Abstract

Problems associated with understanding, verifying and re-engineering the way in which a system allocates and releases dynamic memory present significant challenges to the software maintainer. Because the questions underlying these problems are undecidable, no system can provide a completely fail safe certification. For example, in checking for memory leaks, a system can only warn of potential problems, but cannot guarantee that no leaks remain.

This thesis shows how to denote implicit state components with pseudo variables, making these components explicit. This transformation of an implicit-state program into an explicit-state program is useful because it allows us to form a slicing criterion which captures the (previously implicit) state. The implicit state removal algorithm for a simple intraprocedural language (involving heap allocation using malloc) is presented as a context-free, syntax directed transformation.

Amorphous program slicing is used to create a Dynamic Memory Model (DMM). The slice is constructed from a transformed version of the original program in which heap access is made explicit using a pseudo variable to denote the top of the heap. The DMM is a simplified version of the original program which is concerned solely with dynamic memory access.

The thesis presents 3 case studies to evaluate the use of slicing to construct the DMM of a real program: diff, the GNU differencing program, Word count and DVD stock analysis program. In order to measure the size of a program, the Lines of Code (LoC) metric is used.

The results from various slicing algorithms and approaches need to be comparable using LoC as a metric. This required some pre-processing to ensure that lines of code were counted in a fair and consistent manner and some post-processing to repair the (only slightly) erroneous slices produced by Unravel. The results of the case study show:

- Syntax-preserving static slicing is applicable to the problem of dynamic memory analysis;
- Amorphous static slicing further improves the results of syntax-preserving static slicing;
- Syntax-preserving slicing performs well for Dynamic Memory Modelling.
Acknowledgements

I could not have finished this thesis without the encouragement, help and continues support from my main supervisor Dr Mark Harman of Brunel University. I am so grateful to him for understanding my family situation and allowing me to do the research. He is the best supervisor that I have ever met. In particular I thank him for allowing me to participate in conferences especially on his own fund.

I am also very fortunate to have my second supervisor as Dr Sebastian Danicic of Goldsmiths’ College, London University. He always gives constructive support by making helpful suggestions, criticism and corrections during my research work and preparation of this thesis.

I would like to thank Professor Robert M Hierons of Brunel University for all his help and encouragement, especially for proof reading some of my works.

I would also like to thank the following people:

Professor Dave Binkley of the Loyola College in Maryland for sending helpful comments and suggestions.

Professor Bryan Jones of the University of Derby and Dr Robert Whitrow of the London Metropolitan University for giving their positive feedback and comments.

All the members in the VASTT group for giving me the opportunity to talk about my research work and some positive feedbacks.

Ray Jones of the London Metropolitan University, Professor Dan Simpson of the University of Brighton and Professor Malcolm Munro of the University of Durham for giving me necessary support to complete this thesis.

I would like to express my special thanks to my husband, Siva, and my mum and dad for their support and encouragement throughout the years.

Finally, I would like to thank my daughter, Aarani, for understanding and doing her own works without disturbing me. Also she plays and teaches her sister, Vanee, while I do my research work.
# Contents

1 Introduction ........................................... 11
   1.1 Overview ........................................ 11
   1.2 Main Topic ....................................... 12
   1.3 Syntax-preserving Slicing ......................... 12
   1.4 Amorphous Static Slicing ......................... 13
   1.5 Memory Model .................................. 14
   1.6 Slicing, I/O and the Implicit State ............... 14
   1.7 Analysis of Dynamic Memory Access using Amorphous Slicing ......................... 14
   1.8 Slice-Based Dynamic Memory Modelling — Case Studies ......................... 15
   1.9 Introducing the Pseudo Variable $hp$ ........... 15
   1.10 Contributions of this thesis ..................... 16
   1.11 Thesis Overview ................................ 17

2 Literature Survey .................................. 19
   2.1 Introduction .................................... 19
   2.2 Static Slicing .................................. 20
      2.2.1 Graph Theoretic Definition ................. 20
      2.2.2 Preliminary Definition ...................... 21
      2.2.3 Semantics of a Trajectory ................. 23
      2.2.4 Conditioned Slicing ....................... 25
      2.2.5 Conditioned Slicing Definition .......... 26
   2.3 Dynamic Slicing ................................ 30
   2.4 Amorphous Slicing .............................. 32
### 3 Slicing, I/O and the Implicit State
3.1 Introduction .................................................. 35
3.2 Preliminary Definitions ...................................... 36
3.3 The Formal Definition of a Slice ............................ 36
3.4 The Input Problem ............................................. 38
3.5 Denoting the Input Sequence ................................. 41
3.6 A Thermostat Program ....................................... 44

### 4 Analysis of Dynamic Memory Access using Amorphous Slicing
4.1 Introduction .................................................... 49
4.2 Overview ....................................................... 50
4.3 Modelling the Implicit State ................................ 52
4.4 Amorphous Slicing ........................................... 53
4.5 Domain Specific Transformation ............................. 55
4.6 A Simple Algorithm for Intraprocedural Amorphous
  DMM Slicing .................................................... 56
4.7 Worked Example .............................................. 58
4.8 The Use of DM Ms in Analysis .............................. 64
  4.8.1 Re-Engineering ............................................ 64
  4.8.2 Space Complexity Formulæ ............................. 64
  4.8.3 Identifying Memory Leaks ................................ 66

### 5 Slice–Based Dynamic Memory Modelling — Case Studies
5.1 Introduction .................................................... 69
5.2 Dynamic Memory Modelling ................................ 71
  5.2.1 Syntax–Preserving Slicing .............................. 71
  5.2.2 The Pseudo Variable \( \text{hp} \) .......................... 72
5.3 Preparation for Case Studies ................................ 73
  5.3.1 Assumptions in Counting Lines of Code .............. 74
  5.3.2 Modifying the output of Unravel ....................... 79
5.4 Static Slicing: Results ...................................... 79
5.5 Using Conditioned Slicing .................................. 81
  5.5.1 Conditioned Slicing: Results ......................... 83
5.6 Word count program ........................................ 85
5.7 DVD stock analysis program

6 Conclusions

7 Future Work

A Case Study - diff program and its slice results

A.1 Original diff program
A.2 Unravel version
A.3 After the variable hp introduced
A.4 All cases - Conditioned slice
A.5 All cases - Amorphous slice
A.6 Case F, I, p - Conditioned slice
A.7 Case F, I, p - Amorphous slice
A.8 Case D - Conditioned slice
A.9 Case D - Amorphous slice
A.10 Case x - Conditioned slice
A.11 Case x - Amorphous slice
A.12 Case X - Conditioned slice
A.13 Case X - Amorphous slice
A.14 Case default - Conditioned slice
A.15 Case default - Amorphous slice

B Case Study - Word count program and its slice results

B.1 Word count program
B.2 After the variable hp introduced
B.3 Syntax-preserving slice
B.4 Amorphous slice
Case Study - DVD record count program and its slice results

C.1 DVD record count program
C.2 After the variable hp introduced
C.3 Syntax-preserving slice
C.4 Amorphous slice
# List of Figures

1.1 Example for Syntax-preserving Slice .......................................................... 12
1.2 Example for Amorphous Slice .................................................................. 13
1.3 Example for introducing the Pseudo Variable ......................................... 16
1.4 Amorphous slicing result for Figure 1.3 for the slicing criterion \( h_p, \text{end of program} \) . 16

2.1 Weiser’s static slice .................................................................................. 19
2.2 Static slicing for the criterion \( (5, \{q\}) \) .................................................. 23
2.3 Conditioned slice for criterion \( (7, \{x\}, (5 > n)) \) .............................. 26
2.4 Example for conditioned slice - slicing criterion \( \{\}, (\forall a, a > 4), 7, \{x\} \) .... 27
2.5 Comparison of static and conditioned slicing ........................................... 28
2.6 Program dependence graph for the original program for Figure 2.5 .......... 28
2.7 Simple Example and Different Slices When we Apply Their Definitions .... 29
2.8 Example of a Dynamic Slice ..................................................................... 31
2.9 Static vs. Amorphous Slicing ................................................................... 33

3.1 Slicing Programs with Input Statements .................................................... 38
3.2 An Incorrect Slice .................................................................................... 41
3.3 Removing the Implicit State from a Language with Input Statements ........ 44
3.4 A Simple Thermostat Program .................................................................. 44
3.5 Informal Semantics for the Thermostat Language Primitives ................. 45
3.6 Pseudo-Variables and the Implicit State Components they Denote .......... 45
3.7 Removing the Implicit State from the Thermostat Control Language ....... 46
3.8 Original and Transformed Thermostat Program ...................................... 47
3.9 Slicing the Thermostat Program Using its Explicit Counterpart ............... 47

4.1 A Simple Motivating Example ................................................................. 51
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Making the Heap Top Explicit</td>
</tr>
<tr>
<td>4.3</td>
<td>Transformation Rules for Amorphous DMM Slices</td>
</tr>
<tr>
<td>4.4</td>
<td>The Top Level algorithm</td>
</tr>
<tr>
<td>4.5</td>
<td>Exam Marks Original Program</td>
</tr>
<tr>
<td>4.6</td>
<td>Exam Marks Program</td>
</tr>
<tr>
<td>4.7</td>
<td>Symbolic Execution Algorithm</td>
</tr>
<tr>
<td>4.8</td>
<td>Push Algorithm</td>
</tr>
<tr>
<td>4.9</td>
<td>Using DMMs to Capture Space Complexity of Underlying Algorithms</td>
</tr>
<tr>
<td>4.10</td>
<td>Using DMMs to Identify Memory Leaks</td>
</tr>
<tr>
<td>5.1</td>
<td>Static vs. Amorphous Slicing</td>
</tr>
<tr>
<td>5.2</td>
<td>A Simple DMM Example</td>
</tr>
<tr>
<td>5.3</td>
<td>Example of a special variable declaration</td>
</tr>
<tr>
<td>5.4</td>
<td>Conditional Expression</td>
</tr>
<tr>
<td>5.5</td>
<td>Conditional Statement Rewrite</td>
</tr>
<tr>
<td>5.6</td>
<td>An original complex single statement from diff</td>
</tr>
<tr>
<td>5.7</td>
<td>A complex single statement from diff after re-written</td>
</tr>
<tr>
<td>5.8</td>
<td>LoC for static slicing of diff program</td>
</tr>
<tr>
<td>5.9</td>
<td>Original vs. Static Slices for diff program</td>
</tr>
<tr>
<td>5.10</td>
<td>Comparison of static and conditioned slicing</td>
</tr>
<tr>
<td>5.11</td>
<td>LoC calculation for Conditioned Slicing of diff program</td>
</tr>
<tr>
<td>5.12</td>
<td>Original vs. Conditioned Slicing for diff program</td>
</tr>
<tr>
<td>5.13</td>
<td>Static vs. Conditioned Slicing for diff program</td>
</tr>
<tr>
<td>5.14</td>
<td>LoC for static slicing of Word count program</td>
</tr>
<tr>
<td>5.15</td>
<td>Word count Original vs. Static Slices</td>
</tr>
<tr>
<td>5.16</td>
<td>Comparison of Static and Amorphous slicing for Word count program w.r.t. hp</td>
</tr>
<tr>
<td>5.17</td>
<td>Comparison of Static and Amorphous slicing for DVD stock analysis program w.r.t. hp</td>
</tr>
<tr>
<td>5.18</td>
<td>LoC for static slicing of Word count program</td>
</tr>
<tr>
<td>5.19</td>
<td>DVD Stocking Analysis vs. Static Slices</td>
</tr>
<tr>
<td>7.1</td>
<td>Example of a mark</td>
</tr>
<tr>
<td>7.2</td>
<td>Example of a release</td>
</tr>
<tr>
<td>7.3</td>
<td>Example of a free</td>
</tr>
<tr>
<td>7.4</td>
<td>Example of a realloc</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Overview

Many computer programmers spend a considerable amount of time trying to understand and to manipulate large computer programs [51]. If the program is sufficiently simple, it can be analysed manually. But such a task is too difficult to perform for larger programs, since they contain a lot of irrelevant information for a particular problem analysis of the program.

There are some tools available, such as interactive debuggers, which analyse a program. But these tools give only dynamic information, concerning a single execution. Understanding requires some general information about the program’s behaviour in a variety of situations. One way of solving these problems is by using program slicing [81].

The concept of program slicing was first introduced by Weiser [80]. The process of slicing is to delete unwanted statements from a particular program. It is a useful process that assists the designing of programs more time-effectively.

Program slicing has many applications including

- Program testing [7, 12, 37, 39, 55, 4]
- Program integration [51, 81]
- Parallel program execution [82]
- Program debugging [83, 3, 68, 55, 62, 71]
- Software maintenance [35, 9, 18, 20, 23, 32, 34, 33, 42, 63, 70, 84]
• Program comprehension [8, 28, 44, 45] and
• Software metrics [11, 66, 63, 64, 58, 43, 59].

Tip [76] and Binkley and Gallagher [15] provide detailed surveys of the paradigms, applications and algorithms for program slicing.

The method for reducing the number of statements, which still produce the same behaviour as the original program on specified variable(s), is called *program slicing*. The reduced program which has the advantage of being executable, as well as an independent program, is called a *slice*. This method is generally applicable to procedural languages such as Pascal, C, etc.

1.2 Main Topic

This thesis is concerned with analysing memory leaks in a program using slicing. For the analysis we will primarily focus on Amorphous static slicing [41] but will also consider Conventional Syntax-preserving slicing and Conditioned slicing.

1.3 Syntax-preserving Slicing

Syntax preserving program slicing is a method for deleting commands or statements which has no effect on the set of variables, which are chosen at a particular point or a line in a program. We will be using the term slicing criterion \((V, i)\), which has a set of variables \(V\) and a line number \(i\). Statements which have no effect on \(V\) at \(i\) are removed to form a slice.

Consider the example shown in Figure 1.1

<table>
<thead>
<tr>
<th>Original program</th>
<th>Slice result for the slicing criterion ((x, 4))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: (y = 1;)</td>
<td>1: (y = 1;)</td>
</tr>
<tr>
<td>2: (x = 0;)</td>
<td>3: (x = y + 4;)</td>
</tr>
<tr>
<td>3: (x = y + 4;)</td>
<td></td>
</tr>
<tr>
<td>4: (z = 2x + y;)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.1: Example for Syntax-preserving Slice

In Figure 1.1, the slicing criterion is a single variable \(x\) at end of line 4. To form a slice, line 4 can be deleted because the variable \(z\) is only defined and it does not contribute to the
variable \( x \). Line number 2 can be deleted because \( x \) is re-defined at line number 3 and \( x \) is not referenced itself again. The sliced result is shown on the right side of the Figure 1.1 which is a subset of the original program.

### 1.4 Amorphous Static Slicing

Amorphous static slicing is a variation of syntax-preserving static slicing.

The slice:

1. must preserve the effect of the original program with respect to the slicing criterion.
   
   For example, if an original program gives the result for \( x = 10 \) then the amorphous slice also must give the result 10 for the value of \( x \) when the slicing criterion \( x \) at last line of the program;

2. need not to be a syntactic subset of the original program. See example in Figure 1.2;

3. retains the semantic aspect of syntax-preserving slicing;

4. removes the syntactic restriction to command deletion.

Consider on the same example as we have chosen for syntax-preserving slicing.

<table>
<thead>
<tr>
<th>Original program</th>
<th>Syntax-preserving slice with the slicing criterion ((x,4))</th>
<th>Amorphous slice with the slicing criterion ((x,4))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( y = 1; )</td>
<td>1: ( y = 1; )</td>
<td>3: ( x = 5; )</td>
</tr>
<tr>
<td>2: ( x = 0; )</td>
<td>3: ( x = y + 4; )</td>
<td>4: ( z = 2x + y; )</td>
</tr>
</tbody>
</table>

Figure 1.2: Example for Amorphous Slice

As shown in Figure 1.2, we can apply some transformation rules to the syntax-preserving slice. In this case substituting the value of \( y \) at line 3 gives:

\[ x = (1) + 4; \]

and simplifying the expression gives the amorphous slice. The final value of \( x \) is the same in all three cases. But with amorphous slice, the slice result is not a subset of the original program (i.e. syntactically not a subset).
1.5 Memory Model

The main aim of this thesis is to analyse how much memory is used in a program. In this study we have only focused on a C-like language. In C, allocating memory is used by a function call malloc(). Unfortunately we cannot slice on a function. So we have to make the implicit state explicit. In order to do that we need to introduce a new variable. By doing this, we can slice on the new variable.

1.6 Slicing, I/O and the Implicit State

Some aspects of a program’s semantics are not captured by a set of variables, rendering slicing inapplicable to their analysis. These aspects of the program’s state shall, collectively, be termed the ‘implicit state’. For example, the input list supplied to a program is not denoted by a variable, rather it is part of the implicit state. It will be shown that this implicitness causes existing slicing algorithms to produce incorrect slices with respect to input.

In order to solve this kind of problem, the program should be transformed into an ‘explicit’ version (in which all aspects of its semantics are captured by variables). The approach is also applied to a wider class of problems in which slicing is inhibited by the lack of variables upon which to form a suitable slicing criterion.

1.7 Analysis of Dynamic Memory Access using Amorphous Slicing

The way in which a system allocates and releases dynamic memory indicates significant challenges to the software maintainer. Because the questions underlying these problems are undecidable, no system can provide a completely fail safe certification. For example, in checking for memory leaks, a system can only warn of potential problems, but cannot guarantee that no leaks remain.

This can be addressed by modelling the dynamic memory access properties of a program using amorphous program slicing to create a Dynamic Memory Model (DMM). The slices can
be constructed from a transformed version of the original program in which heap access has been made explicit using a pseudo variable to denote the top of the heap. The DMM is a simplified version of the original program which is concerned solely with the dynamic memory access behaviour of the original.

This thesis will construct proof of concept DMM construction slicing algorithm for a heap using C-like language. This will require to show how slicing simplification power can be dramatically improved using domain-specific transformation rules.

1.8 Slice-Based Dynamic Memory Modelling — Case Studies

Slices can be constructed statically (with respect to no input information) or conditionally (with respect to partial input information). They can also be constructed in either a purely syntax-preserving or amorphous way. Amorphous slices tend to be smaller than their syntax-preserving counterparts, but they may not be syntactically related to the original.

One of the case study [72] is conducted using a real life program, such as diff program to assess the value of static and conditioned slicing (in both syntax-preserving and amorphous formulations) upon the problem of dynamic memory analysis.

The Word count program shows that there is no memory leaks at all. On the other hand, the DVD stock analysis program clearly illustrates how much memory still has not been released.

1.9 Introducing the Pseudo Variable hp

We chose the new pseudo variable to be hp. The variable hp will always be declared as a global variable which is not shown in Figure 1.3. In order to use this variable, first we need to initialise it to zero. Every time when the malloc() function is used, we need to increase the value of the variable hp by the amount of memory that gets allocated.
In Figure 1.4 when we amorphously slice on variable hp at end of the program we will get the result shown on right side of Figure 1.4. This means the while loop get executed $N$ number of times. Therefore, the allocated memory would be $N$ number of times of `sizeof(struct L)`. Some examples and detail explanation are given in Chapter 4.

### 1.10 Contributions of this thesis

The contributions of this thesis are as follows:
This thesis is concerned with analysing memory leaks in a program using slicing. For the analysis we will primarily focus on Amorphous static slicing. We will also consider Conventional Syntax-preserving slicing and Conditioned slicing.

This thesis shows how to denote implicit state components with pseudo variables, making these components explicit.

The work is evaluated with 3 case studies and has examined the effect of syntax-preserving slicing and its amorphous counter-part in the analysis of dynamic memory allocation for the open source GNU diff program, Word count and DVD stock analysis program.

Both static and conditioned forms of slices were constructed for the diff program. The latter being constructed (partly) by hand (because conditioned slicing technology is still in its infancy, and therefore incapable of slicing a program, the size of diff).

1.11 Thesis Overview

The contribution of each chapters can be summarised as follows:

• Chapter 2: Literature Survey

This chapter introduces the basic concept of program slicing and various different slicing methods such as Static, Conditioned, Dynamic and Amorphous slicing. Also various other definitions are explained.

• Chapter 3: Slicing, I/O and the Implicit State

This chapter is concerned with the kinds of slices constructed from programs which perform I/O, and, more generally with slices of programs which affect components of the state for which there is no variable to capture the semantic projection of interest.

• Chapter 4: Analysis of Dynamic Memory Access using Amorphous Slicing

This chapter presents an approach to modelling the dynamic memory access properties of a program using amorphous program slicing to create a Dynamic Memory Model.
Chapter 4 also introduces a proof-of-concept Dynamic Memory Model construction algorithm, showing how slicing simplification power can be dramatically improved using domain-specific transformation rules.

- Chapter 5: Slice-Based Dynamic Memory Modelling — Case Studies
  This chapter provides a case study of a real program: diff, the GNU differencing program, Word count and DVD stock analysis program. It also explores the results of the case studies which assesses the value of static and conditioned slicing (in both syntax-preserving and amorphous formulations) upon the problem of dynamic memory analysis.

- Chapter 6: Conclusions
  This chapter shows the conclusions from the work described in this thesis.

- Chapter 7: Future Work
  This chapter presents some extensions, problems and possible future work.

- Appendices A, B and C: Case Studies - Example programs and their slice results
  These appendices contain complete source code for the diff, Word count and DVD stock analysis programs which are used in chapter 5 and the results for the case studies.
Chapter 2

Literature Survey

2.1 Introduction

In this chapter, various different slicing methods are described. These include Static, Conditioned, Dynamic and Amorphous slicing.

As introduced by Weiser[82], slicing is carried out at line number $n$, with respect to a given set of variables $S$. Consider the following example program in Figure 2.1:

```
1 z = 4;
2 y = z + 1;
3 x = 5 + z;
4
```

Original Program

```
1 z = 4;
2 y = z + 1;
3 x = 5 + z;
4
```

Slice w.r.t. ($\{ x \}$, 4)

Figure 2.1: Weiser’s static slice

The selected variable was $x$ and the slicing was performed at line 4, i.e. the pair ($\{ x \}$, 4), is known as the slicing criterion. In this chapter, we use the term slice set to refer to the set $\{ x \}$. The idea of slicing is to delete unwanted statements from the program. In Figure 2.1, the variable $x$ does not depends on $y$, hence the slice program does not contain line 2. It is a fairly simple operation, but it is often difficult to make a precise decision as to what lines can be deleted.
There can be many difference slices for a given program depending upon the choice of slicing criterion. For any slicing criterion it is important for the slice of a program to maintain the effect of the original program upon the slice set.

Normally sliced programs are thinner than the original program, but there are some occasions when the original program itself could be the sliced program. For example, consider the example program in Figure 2.1. Suppose the slicing criterion is (\{ x, y \}, 4), then the sliced program would be the original program itself. More recently, several variation on the slicing theme has been introduced.

2.2 Static Slicing

The first published definition of a program slicing was given by Mark Weiser, who introduced the concept of program slicing in his 1979 doctoral thesis[80]. A static slice is constructed with respect to the pair \(<i, V>\), where \(i\) denotes a specific statement in a program and \(V\) is a subset of variables in that program.

2.2.1 Graph Theoretic Definition

The definition of digraph, path, flowgraph, hammock graph, state trajectory, static slicing criterion, projection and static slicing are taken from Weiser[82].

Definition 2.2.1 (Digraph)

A digraph is a structure \(<N, E>\), where \(N\) is a set of nodes and \(E\) is a set of edges in \(N \times N\). If \((n, m)\) is \(\in E\), then \(n\) is an immediate predecessor of \(m\) and \(m\) is an immediate successor of \(n\).

Definition 2.2.2 (Path)

A path from \(n\) to \(m\) of length \(k\) is a list of nodes \(p_0, p_1, \ldots, p_k\) such that \(p_0 = n, p_k = m\), and for all \(i, 1 \leq i \leq k - 1, (p_i, p_{i+1})\) is in \(E\).
Definition 2.2.3 (Flowgraph)
A flowgraph is a structure $<N, E, n_0>$, where $<N, E>$ is a digraph and $n_0$ is a member of $N$ such that there is a path from $n_0$ to all other nodes in $N$. $n_0$ is sometimes called the initial node. If $m$ and $n$ are two nodes in $N$, $m$ dominates $n$ if $m$ is on every path from $n_0$ to $n$.

Definition 2.2.4 (A Hammock Graph)
A hammock graph is a structure $<N, E, n_0, n_e>$ with the property that $<N, E, n_0>$ and $<N, E^{-1}, n_e>$ are both flowgraphs. Note that, as usual, $E^{-1} = \{(a, b) \mid (a, b) \text{ is in } E\}$. If $m$ and $n$ are two nodes in $N$, $m$ inverse dominates $n$ if $m$ is on every path from $n$ to $n_e$.

Hammock graph is single-entry single-exit.

In the next definition, the flowgraph is assumed to be hammock graph and reference and defined variables are formally introduced.

2.2.2 Preliminary Definition

Definition 2.2.5 (Reference and Defined Variables)
Let $V$ be the set of variable names which appear in a program $P$. Then for each statement $n$ in $P$ (i.e., node in the flowgraph of $P$) will have the following two sets, each a subset of $V$:
- $REF(n)$ is the set of variables whose values are used at $n$, and
- $DEF(n)$ is the set of variables whose values are changes at $n$.

A state trajectory of a program is a trace of its execution, containing 'snapshots' of all its variable values (its state) just before the execution of each statement.

Definition 2.2.6 (State Trajectory) A state trajectory of length $k$ of a program $P$ is a finite list of ordered pairs

$$\langle (n_1, s_1), (n_2, s_2), \ldots, (n_k, s_k) \rangle$$

where each $n$ is a node of $P$ and each $s$ is a function mapping the variables in $V$ to their values. Each $(n, s)$ gives the values of variables in $V$ immediately before the execution of $n$. 
**Definition 2.2.7 (Static Slicing Criterion)** A *static slicing criterion* of a program $P$ is a pair $<i, V>$, where $i$ is a statement in $P$ and $V$ is a subset of the variables in $P$.

A static slicing criterion $C = <i, V>$ determines a projection function $\text{Proj}_C$ which throws out of the state trajectory all ordered pairs except those starting with $i$, and from the remaining pairs throws out all identifiers not in $V$.

**Definition 2.2.8 (Projection)**

Let $T = (t_1, t_2, \ldots, t_n)$ be a state trajectory, $a$ any node in $N$ and $s$ any function from variable names to values. Then

$$\text{Proj}_{<i, V>}((n, s)) = \begin{cases} 
\lambda & \text{if } n \neq i \\
(n, s|V) & \text{if } n = i
\end{cases}$$

where $s|V$ is $s$ restricted to domain $V$, and $\lambda$ is the empty string. $\text{Proj}'$ is now extended to entire trajectories:

$$\text{Proj}_{<i, V>}(T) = \text{Proj}'_{<i, V>}((t_1)) \ldots \text{Proj}'_{<i, V>}((t_n)).$$

A slice is defined, behaviourally, as any subset of a program which preserves a projection of its behaviour, determined by the slicing criterion.

**Definition 2.2.9 (Static Slicing)** A slice $S$ of a program $P$ on a slicing criterion $C = <i, V>$ is any program with the following two properties.

1. $S$ can be obtained from $P$ by deleting zero or more statements from $P$.

2. Whenever $P$ halts on an input $I$ with state trajectory $T$, then $S$ also halts on input $I$ with state trajectory $T'$, and $\text{Proj}_C(T) = \text{Proj}_{C'}(T')$, where $C' = <\text{succ}(i), V>$, and $\text{succ}(i)$ is the nearest successor to $i$ in the original program which is also in the slice, or $i$ itself if $i$ is in the slice.
**Definition 2.2.10 (Tuple)**

For a given tuple^1^ \( t \) and a number \( n \), we can say that \( t \mid n \) means the \( n \)th element of tuple \( t \).

Let \( t = (2, x, 5) \) i.e. \( t \) is a 3-tuple then

- \( t \mid 1 = 2 \)
- \( t \mid 2 = x \)
- \( t \mid 3 = 5 \)

\( t \mid n \) is undefined if \( (t < 1) \) or \( (t > 3) \)

Now we can write \( (T(\#T) \mid 2)([x]) = 4 \) for the example program in Figure 2.2.

**Definition 2.2.11 (Head and Tail)**

The head of a sequence, \( s \), shall be denoted \( \text{hd}(s) \) and the remaining sequence shall be denoted \( \text{tl}(s) \). We shall consider the following list:

\[
\begin{align*}
1 &= [1, 2, 3, 4, 5] \\
\text{hd}(l) &= 1 \\
\text{tl}(l) &= [2, 3, 4, 5]
\end{align*}
\]

**Definition 2.2.12 (Function Overriding)**

This is an operation that takes two functions and creates a new one by overriding all the arrows in the first function which those in the second. More formally we write the function overriding of the function, \( f \), with the function \( g \) like this: \( f \oplus g \).

**2.2.3 Semantics of a Trajectory**

We will assume the initial state for program \( P \) is \( \sigma \) (example program in Figure 2.2).

^1^ordered structures, in our case a pair is known as a 2-tuple.
The state \( \sigma \) referred to in a trajectory occurs in a pair \((n, \sigma)\) where \(n\) is a next line to be executed, therefore at line 1, the state is still at initial state (i.e. \((1, \sigma)\)). The term ‘at line \(n\)’ thus means ‘when the next line to be executed is at line \(n\).’

After executing line 1, the variable \(x\) will have the value 4, therefore at line 2 the state is as follows:

\[
(2, \sigma \oplus \{([x] \leftarrow 4)\})
\]

At line 3, \(z\) will have the value 8 and the state is:

\[
(3, \sigma \oplus \{([x] \leftarrow 4), ([z] \leftarrow 8)\})
\]

Since \(x\) has the value 4 in the state at line 3, \(q\) will have the value 4 after executing line 3 because it is argued from \(x\), therefore the state is:

\[
(4, \sigma \oplus \{([x] \leftarrow 4), ([z] \leftarrow 8), ([q] \leftarrow 4)\})
\]

At line 5, the variable \(y\) will have the value 12 and the state is as follows:

\[
(5, \sigma \oplus \{([x] \leftarrow 4), ([z] \leftarrow 8), ([q] \leftarrow 4), ([y] \leftarrow 12)\})
\]

The state trajectory \(T\) for \(P\) is:

\[
T = (1, \sigma),
(2, \sigma \oplus \{([x] \leftarrow 4)\}),
(3, \sigma \oplus \{([x] \leftarrow 4), ([z] \leftarrow 8)\}),
(4, \sigma \oplus \{([x] \leftarrow 4), ([z] \leftarrow 8), ([q] \leftarrow 4)\}),
(5, \sigma \oplus \{([x] \leftarrow 4), ([z] \leftarrow 8), ([q] \leftarrow 4), ([y] \leftarrow 12)\})
\]

Suppose we want to find the final value of a variable for the above example in Figure 2.2, then it must be found from the final state of that program. i.e.

\[
(5, \sigma \oplus \{([x] \leftarrow 4), ([z] \leftarrow 8), ([q] \leftarrow 4), ([y] \leftarrow 12)\})
\]
Let $T(\#T) = T(5)$

$$= (5, \sigma \oplus \{(x \mapsto 4), (z \mapsto 8), (\varphi \mapsto 4), (y \mapsto 12)\})$$

then the final value of $x$ is

$$(5, \sigma \oplus \{(x \mapsto 4), (z \mapsto 8), (\varphi \mapsto 4), (y \mapsto 12)\})$$

Therefore $\text{Proj}_{<5, q>}(T) = \text{Proj}_{<5, q>}(T')$.

The state trajectory $T'$ for the sliced program $P'$ when the initial state is $\sigma$ is given below:

$$T' = \langle (1, \sigma),
(2, \sigma \oplus \{(x \mapsto 4)\}),
(3, \sigma \oplus \{(x \mapsto 4)\}),
(4, \sigma \oplus \{(x \mapsto 4), (\varphi \mapsto 4)\}),
(5, \sigma \oplus \{(x \mapsto 4), (\varphi \mapsto 4)\}) \rangle$$

Now $\text{Proj}_{<5, q>}(T') = \langle (5, \{(\varphi \mapsto 4)\}) \rangle$

Therefore $\text{Proj}_{<5, q>}(T) = \text{Proj}_{<5, q>}(T')$, therefore $P'$ is a slice of $P$ according to Weiser's definition 2.2.9.

2.2.4 Conditioned Slicing

Canfora, Cimitile, De Lucia and Di Luca are the first people to introduce Conditioned Slicing. The following definitions are taken from[20].

In here they have defined the slicing criterion which is same as Weiser’s definition 2.2.7.

Definition 2.2.13 (Conditioned Slicing Criterion)

A conditioned slicing criterion of a program $P$ is a triple $\langle i, V, C \rangle$ where $i$ is a statement in $P$, $V$ is a subset of the variables in $P$ and $C$ is a condition.
Definition 2.2.14 (Conditioned Slice)

A conditioned slice of a program $P$ on a conditioned slicing criterion $<i, V, C>$ consists of all the statements and predicates of $P$ that might affect the values of the variables in $V$ just before the statement $i$ is executed, when the condition $C$ holds true.

```
1: n = 4;
2: scanf("%d", a);
3: if (a > n)
   4:   x = a - n;
    else
   5:   x = a + n;
6: n = x + a;
7: 
```

<table>
<thead>
<tr>
<th>Original Program</th>
<th>Conditioned Slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: n = 4;</td>
<td>1: n = 4;</td>
</tr>
<tr>
<td>2: scanf(&quot;%d&quot;, a);</td>
<td>2: scanf(&quot;%d&quot;, a);</td>
</tr>
<tr>
<td>3: if (a &gt; n)</td>
<td>3: if (a &gt; n)</td>
</tr>
<tr>
<td>4: x = a - n;</td>
<td>4: x = a - n;</td>
</tr>
<tr>
<td>5: x = a + n;</td>
<td>5:</td>
</tr>
<tr>
<td>6: n = x + a;</td>
<td>6:</td>
</tr>
<tr>
<td>7:</td>
<td>7:</td>
</tr>
</tbody>
</table>

Figure 2.3: Conditioned slice for criterion $(7, \{x\}, (5 > n))$

The condition $C$ is a boolean expression and it can only consider one input to variable $a$ for the above example in Figure 2.3, which is $a = 5$. In order to overcome this condition $C$ as a boolean expression De Lucia et al. [28], the condition is modified to accommodate with the first order logic formula, which is explained in section 2.2.5.

2.2.5 Conditioned Slicing Definition

The following definitions, 2.2.15, 2.2.16 and 2.2.17 are taken from[28].

Definition 2.2.15 (Conditioned Slicing Criterion)

Let $V_{in}$ be the set of input variables of a program $P$, $V'_{in} \subseteq V_{in}$ and $F'$ be a first order logic formula on the variables in $V'_{in}$. A conditioned slicing criterion of a program $P$ is a quadruple $C = (V'_{in}, F', p, V)$, where $p$ is a statement in $P$ and $V$ is a subset of the variables in $P$.

The formula $F'$ identifies a set of input to the program and consequently a set of state trajectories.

Definition 2.2.16 (Input Set)

Let $V_{in}$ be the set of input variables of a program $P$, $V'_{in} \subseteq V_{in}$ and $F'$ a first order logic
2.2 Static Slicing

formula on the variables in $V'_\text{in}$. Let $IS'(F)$ be the set of input $I'$ for $V'_\text{in}$ that satisfies the formula $F'$. The input set $IS(F)$ of $P$ with respect to $F$ is the set of input $I$ to the program such that $I$ is a completion of some $I' \in IS'(F)$.

Each $I \in IS(F)$ identifies a trajectory $T$. A conditioned slice is any subset of the program which reproduces the original behaviour on each of these trajectories.

**Definition 2.2.17 (Conditioned Slice)**

A conditioned slice of a program $P$ on a conditioned slicing criterion $C = (V'_\text{in}, F, p, V)$ is any syntactically correct and executable program $P'$ that is obtained from $P$ by deleting zero or more statements, and whenever $P$ halts on input $I$, $I \in IS(F)$, with state trajectory $T$, then $P'$ also halts on input $I$ with state trajectory $T'$, and $\text{Proj}_{p,V}(T) = \text{Proj}_{p,V}(T')$.

```plaintext
1: n = 4;
2: scanf("%d", a);
3: if (a > n)
4:   x = a - n;
    else
5:   x = a + n;
6: n = x + a;
7: 

Original Program

1: n = 4;
2: scanf("%d", a);
3: if (a > n)
4:   x = a - n;
5: 
6: 
7: 

Conditioned Slice
```

Figure 2.4: Example for conditioned slice - slicing criterion ($\emptyset, (\forall a, a > 4), 7, \{x\}$)

As shown in Figure 2.4, the slicing criterion is ($\emptyset, (\forall a, a > 4), 7, \{x\}$) and the Conditioned slice result does not contain an else part. This is due to the condition, which will be always true for all values of $a$ for greater than 4.

In order to compare the differences between the conditioned slice and Weiser's slice, we need to apply De Lucia *et al* algorithm to do the slicing. A simple example is given in Figure 2.5.
1: \text{Max} = 10; \\
2: \text{Min} = 0; \\
3: \text{if (Max < Min)} \\
4: \quad x = \text{Max}; \\
\quad \text{else} \\
5: \quad x = \text{Min}; \\
6: \quad y = \text{Max} + \text{Min}; \\
7: \\
\begin{array}{|c|c|c|}
\hline
\text{Original Program} & \text{Conditioned Slice for the Slicing Criterion} & \text{Weiser’s Slice for the Slicing Criterion} \\
\hline
1: \text{Max} = 10; & 1: \text{Max} = 10; & 1: \text{Max} = 10; \\
2: \text{Min} = 0; & 2: \text{Min} = 0; & 2: \text{Min} = 0; \\
3: \text{if (Max < Min)} & 3: \text{if (Max < Min)} & 3: \text{if (Max < Min)} \\
4: \quad x = \text{Max}; & 4: \quad x = \text{Max}; & 4: \quad x = \text{Max}; \\
\quad \text{else} & \quad \text{else} & \quad \text{else} \\
5: \quad x = \text{Min}; & 5: \quad x = \text{Min}; & 5: \quad x = \text{Min}; \\
6: \quad y = \text{Max} + \text{Min}; & 6: \quad y = \text{Max} + \text{Min}; & 6: \quad y = \text{Max} + \text{Min}; \\
7: & 7: & 7: \\
\hline
\end{array}

Figure 2.5: Comparison of static and conditioned slicing

Also, program dependence graph [53] is shown in Figure 2.6 for the original program. Thick lines shows the Control dependencies and thin lines shows Data dependencies.

Figure 2.6: Program dependence graph for the original program for Figure 2.5

But if we use their definitions we will get the following slice, which shows in Figure 2.7.
### 2.2 Static Slicing

<table>
<thead>
<tr>
<th>Original Program</th>
<th>Conditioned Slice for the Slicing Criterion ((\emptyset, \text{True}, 7, {x}))</th>
<th>Weiser's Slice for the Slicing Criterion (({x}, 7))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1: \text{Max} = 10;)</td>
<td>(1: \text{Min} = 0;)</td>
<td>(1: \text{Min} = 0;)</td>
</tr>
<tr>
<td>(2: \text{Min} = 0;)</td>
<td>(2: \text{Min} = 0;)</td>
<td>(2: \text{Min} = 0;)</td>
</tr>
<tr>
<td>(3: \text{if} (\text{Max} &lt; \text{Min}))</td>
<td>(3: \text{if} (\text{Max} &lt; \text{Min}))</td>
<td>(3: \text{if} (\text{Max} &lt; \text{Min}))</td>
</tr>
<tr>
<td>(4: \text{x} = \text{Max};)</td>
<td>(4: \text{x} = \text{Max};)</td>
<td>(4: \text{x} = \text{Max};)</td>
</tr>
<tr>
<td>(\text{else})</td>
<td>(\text{else})</td>
<td>(\text{else})</td>
</tr>
<tr>
<td>(5: \text{x} = \text{Min};)</td>
<td>(5: \text{x} = \text{Min};)</td>
<td>(5: \text{x} = \text{Min};)</td>
</tr>
<tr>
<td>(6: \text{y} = \text{Max} + \text{Min};)</td>
<td>(6: \text{y} = \text{Max} + \text{Min};)</td>
<td>(6: \text{y} = \text{Max} + \text{Min};)</td>
</tr>
<tr>
<td>(7: )</td>
<td>(7: )</td>
<td>(7: )</td>
</tr>
</tbody>
</table>

Figure 2.7: Simple Example and Different Slices When we Apply Their Definitions

We will assume the initial state for the Original Program is \(\sigma\). The state trajectory \(T\) for the Original Program is:

\[
T = < (1, \sigma), \]
\[
(2, \sigma \oplus \{([\text{Max}] \mapsto 10)\}), \]
\[
(3, \sigma \oplus \{([\text{Max}] \mapsto 10), ([\text{Min}] \mapsto 0)\}), \]
\[
(5, \sigma \oplus \{([\text{Max}] \mapsto 10), ([\text{Min}] \mapsto 0)\}), \]
\[
(6, \sigma \oplus \{([\text{Max}] \mapsto 10), ([\text{Min}] \mapsto 0), ([x] \mapsto 0)\}), \]
\[
(7, \sigma \oplus \{([\text{Max}] \mapsto 10), ([\text{Min}] \mapsto 0), ([x] \mapsto 0), ([y] \mapsto 10)\}) \>
\]

\[
\text{Proj}_{1,\sigma}\rangle(T) = < (7, \{([x] \mapsto 0)\}) \rangle
\]

The state trajectory \(T'\) for Conditioned slice and the Weiser's slice is:

\[
T' = < (1, \sigma), \]
\[
(2, \sigma), \]
\[
(3, \sigma \oplus \{([\text{Min}] \mapsto 0)\}), \]
\[
(4, \sigma \oplus \{([\text{Min}] \mapsto 0)\}), \]
\[
(5, \sigma \oplus \{([\text{Min}] \mapsto 0)\}), \]
\[
(6, \sigma \oplus \{([\text{Min}] \mapsto 0), ([x] \mapsto 0)\}), \]
\[
(7, \sigma \oplus \{([\text{Min}] \mapsto 0), ([x] \mapsto 0)\}) \>
\]

\[
\text{Proj}_{1,\sigma}\rangle(T') = < (7, \{([x] \mapsto 0)\}) \rangle
\]
So according to Weiser’s definition 2.2.9 and De Lucia’s definition 2.2.17
\[
\text{Proj}_{\xi,x}(T) = \text{Proj}_{\xi,x}(T') = \langle 7, \{x \mapsto 0\} \rangle
\]

2.3 Dynamic Slicing

Static slices have to preserve a projection of the semantics of the original program for every possible execution of the program. This tends to yield rather large slices for most well written programs which are highly cohesive. This observation was the motivation for work on the use of slicing as the basis for the calculation of measurements of program cohesion based upon the construction of program slices [11, 65, 66, 60, 63, 74, 40].

The original inputs behind the development of program slicing arose because of the use of slices in bug-location. This motivation for slicing, combined with the rather disappointingly large slices which were constructed from static slicing algorithms lead to interest in dynamic slicing.

Korel and Laski [56] introduced the concept of a dynamic slice, constructed with respect to an input sequence \(x\). A dynamic slice need only preserve the effect of the original program upon the slicing criterion when supplied with input \(x\). The dynamic paradigm is ideally suited to the bug-location application, because a bug is usually detected as the result of the execution of a program with respect to some specific input.

The example in Figure 2.8 illustrate when the input value for \(x = -1\) at the end of the program.
A pure definition of dynamic slicing (so-named for reasons which will become clear shortly) is given in definition 2.3.1 below.

**Definition 2.3.1 (Pure Dynamic Slice)** A dynamic backward slice of a program \( p \) is constructed with respect to a slicing criterion \( (V, n, x) \), where \( V \) is a set of variables, \( n \) is a point of interest within \( p \) and \( x \) is an input sequence. A dynamic backward slice preserves the projected meaning of \( p \) with respect to \( V \) at \( n \) when supplied with the input \( x \).

Definition 2.3.1 is *not* the definition put forward by Korel and Laski. The original definition of a dynamic slice, introduced by Korel and Laski and called a KL-slice in this thesis, is given in definition 2.3.2 below:

**Definition 2.3.2 (KL slice)** Let \( C = (x, I^i, V) \) be a slicing criterion of a program \( P \) and \( T \) a trajectory of \( P \) on input \( x \). A KL dynamic slice of \( P \) on \( C \) is any executable program \( P' \) that is obtained from \( P \) by deleting zero or more statements from it and, when executed on input \( x \), produces a trajectory \( T' \) for which there exists an execution position \( q' \) such that:

1. \( F(T', q') = DEL(F(T, q), T(i) \not\in N' \land 1 \leq i \leq q) \)
2. for all \( v \in V \), the value of \( v \) before the execution of instruction \( T(q) \) in \( T \) equals the value of \( v \) before the execution of instruction \( T'(q') \) in \( T' \)
3. \( T'(q') = T(q) = 1 \)
where $N'$ is a set of instructions in $P'$.

The definition uses two auxiliary functions on sequences, $F$ and $DEL$. $F(T, i)$ is the ‘front’ $i$ elements of $T$ from 1 to $i$ inclusive. $DEL(T, \pi)$ is a filtering operation, which takes a predicate $\pi$ and returns the sequence obtained by deleting elements of $T$ which satisfy $\pi$.

Definition 2.3.2 is constructed for a point of interest, $n$, not in the program, but in the trace of the program induced by the input $x$. This means that the definition also introduces the independent concept of an iteration count for which a slice is constructed.

### 2.4 Amorphous Slicing

Syntax-preserving slicing consists of deleting statements which can be determined to have no effect upon the slicing criterion. This leads to a slice being a syntactic subset of the original from which it is constructed.

**Definition 2.4.1 (Amorphous Static Slicing)**

An amorphous slice, $s$ of a program $p$ is constructed with respect to a slicing criterion, $(V, n)$.

**(semantic property)**

For all variables $v$ in $V$, and for all initial states, the value of $v$ at point $n$ in $s$ must be identical to the value of $v$ at point $n$ in $p$.

**(syntactic property)**

An amorphous slice is constructed with respect to a simplicity measure$^2$, $\subseteq$, which orders programs according to some notion of simplicity.

$s$ is an amorphous slice of $p$ iff it respects the semantic property and $s \subseteq p$.

Amorphous slicing [16, 14, 41, 79] is a variation of syntax-preserving slicing in which the slice must preserve the effect of the original program with respect to the slicing criterion, whilst it need not be a syntactic subset of the original.

---

$^2$In this thesis we shall use a node count, that is

$$p \subseteq q \iff \#(\text{dom(CFG}(p))) \leq \#(\text{dom(CFG}(p)))$$

where $\#$ is set cardinality and $\text{dom(CFG}(x))$ is the domain of the CFG of the program $x$. 
Amorphous slicing therefore retains the semantic aspect of syntax-preserving slicing, but removes the syntactic restriction to command deletion. Clearly, a syntax-preserving slice is also an amorphous slice but not vice-versa. As a consequence, amorphous slices are always no larger than their syntax-preserving counterparts and are often considerably smaller.

Consider the example fragment in Figure 2.9. Column (d) of the figure, contains the amorphous slice for the variable \( r \). The amorphous slice is smaller than the syntax-preserving slice (which is the entire program), but it does not preserve the syntax of the original.

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>Slice for ( x )</td>
<td>Amorphous slice for ( x )</td>
<td>Amorphous slice for ( r )</td>
</tr>
<tr>
<td>( z = 4 )</td>
<td>( z = 4 )</td>
<td>( x = 9 )</td>
<td>( r = 14 )</td>
</tr>
<tr>
<td>( y = z + 1 )</td>
<td>( x = 5 + z )</td>
<td>( x = 5 + z )</td>
<td></td>
</tr>
<tr>
<td>( x = 5 + z )</td>
<td>( r = x + y )</td>
<td>( r = x + y )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.9: Static vs. Amorphous Slicing
Chapter 3

Slicing, I/O and the Implicit State

3.1 Introduction

Some aspects of a program’s semantics are not captured by a set of variables, rendering slicing inapplicable to their analysis. These aspects of the program’s state shall, collectively, be termed the ‘implicit state’. For example, the input list supplied to a program is not denoted by a variable, rather it is part of the implicit state. It will be shown that this implicitness causes existing slicing algorithms to produce incorrect slices with respect to input.

In order to solve the problem the program to be sliced will be transformed into an ‘explicit’ version (in which all aspects of its semantics are captured by variables). The approach is also applied to a wider class of problems in which slicing is inhibited by the lack of variables upon which to form a suitable slicing criterion.

Because the approach can be expressed as a source-level transformation, it has the attractive property that the slicing algorithm need not be altered.

In this chapter we are mainly concerned with the kinds of slices constructed from programs, which perform Input/Output. We especially focus on programs, where there is no variable to capture the semantics projection of interest. In order to study this problem the static slicing paradigm will be adopted for simplicity of exposition. However, the results apply equally well to the dynamic [56], quasi-dynamic [77] and conditioned [20, 28] paradigms.

The contribution of this chapter can be summarised as follows:
A minor problem concerning slicing in the presence of input is identified.

The problem is circumvented using an implicit state removal transformation.

The transformation is shown to be applicable to a wider class of programs which contain few (or no) variables upon which to form slicing criteria.

The rest of the chapter is organised as follows: Section 3.2 contains some preliminary definitions, which are used in sections 3.3 and 3.4 to provide a formal treatment of static slicing in the presence of input statements. Section 3.5 introduces the implicit state removal transformation, used to rectify a problem identified with slicing in the presence of input, and section 3.6 shows how this approach can be applied to the more general problem of slicing real-time system programs, which may contain few variables upon which to base a suitable slicing criterion.

### 3.2 Preliminary Definitions

This section introduces some definitions which will be used in subsequent sections. The other definitions are defined in Chapter 2 - literature survey.

**Definition 3.2.1 (REF and DEF Variable Sets)**

$DEF(n)$ denotes the set of variables defined at node $n$. $REF(n)$ denotes the set of variables referenced at node $n$. For example, if $n$ were the assignment statement $x = y + z$; we would have $DEF(n) = \{x\}$ and $REF(n) = \{y, z\}$.

### 3.3 The Formal Definition of a Slice

Suppose the initial state for program $P$ in Figure 2.2 is $\sigma$. The elements of the state trajectory are pairs, $(n, \sigma)$ where $n$ is the next line to be executed. Therefore at line 1, the pair will be $(1, \sigma)$. The term ‘at line $n$’ means ‘when the next line to be executed is at line $n$'.
3.3 The Formal Definition of a Slice

In order to define the state trajectory produced by the execution of a program, it will be necessary to formally define the state-to-state mapping\(^1\), \(M_I[s]\), denoted by an assignment statement, \(s\). This is defined in the standard way \([73]\), namely:

\[
M_I[i := e] = \lambda \sigma. \sigma \oplus \{ i \leftarrow \mathcal{E}[e] \sigma \}
\]

where \(M_I\) is a mapping from statements to state to state mappings and where a state is a mapping from identifiers to the values they contain.

Using this semantic description, the state trajectory \(T\), for the example program \(P\) can be determined:

\[
T = <\langle 1, \sigma \rangle, \\
\langle 2, \sigma \oplus \{[x] \leftarrow 4\} \rangle, \\
\langle 3, \sigma \oplus \{[x] \leftarrow 4, [z] \leftarrow 8\} \rangle, \\
\langle 4, \sigma \oplus \{[x] \leftarrow 4, [z] \leftarrow 8, [q] \leftarrow 4\} \rangle, \\
\langle 5, \sigma \oplus \{[x] \leftarrow 4, [z] \leftarrow 8, [q] \leftarrow 4, [y] \leftarrow 12\} \rangle >
\]

\(\text{Proj}_{<n,V>}\) denotes the sequence obtained by removing all pairs \((x,y)\) such that \(x \neq n\), and restricting the state, \(y\), of those which remain, to include only those mappings for variables in \(V\), so \(\text{Proj}_{<5,(q)>}(T) = <(5, \{[q] \leftarrow 4\})>\)

The state trajectory \(T'\) for the slice, \(P'\), when the initial state is \(\sigma\), is:

\(^1\)The reason we add the subscript \(I\) to \(M\) will become clear in the next two sections.
T' =
<
(1, σ),
(2, σ ⊕ {x} ← 4),
(3, σ ⊕ {x} ← 4),
(4, σ ⊕ {x} ← 4, [q] ← 4),
(5, σ ⊕ {x} ← 4, [q] ← 4)
>
Now \( \text{Proj}_{<5,(q)}> (T') = <(5, \{[q] ← 4\})> \), so \( \text{Proj}_{<5,(q)}> (T) = \text{Proj}_{<5,(q)}> (T') \), and therefore \( P' \) is a slice of \( P \) according to definition 2.2.9.

3.4 The Input Problem

In the previous section, the state was described as a mapping, \( I \rightarrow V \), where \( I \) is the set of all variable identifiers and \( V \) is the set of all possible values. This form of state is known as an environment and \( V \) is known as the set of denotable values [73].

In order to represent the semantics of input statements we shall need to augment the environment with a sequence of values, \( \text{seq}(V) \), to denote the input sequence, thus the state will become \( (I \rightarrow V) \times \text{seq}(V) \). This augmentation of the state allows us to model the statement \( \text{scanf}("\%d", &x) \); as a state transformation from \( (σ, i) \) to \( (σ \oplus \{[x] \leftarrow \text{hd}(i)\}, \text{tl}(i)) \), enabling us to construct state trajectories for programs which perform input.

Consider, for example the program \( P \) in Figure 3.1. Suppose the slicing criterion is \( (3, \{y\}) \). The state trajectory, \( T \), when the initial environment is \( σ \) and the initial input list is \( i \) is:
3.4 The Input Problem

\[ T = \]
\[ < \]
\[ (1, (\sigma, i)), \]
\[ (2, (\sigma \oplus \{ [x] \mapsto \text{hd}(i) \}, \text{tl}(i))), \]
\[ (3, (\sigma \oplus \{ [x] \mapsto \text{hd}(i), [y] \mapsto \text{hd}(\text{tl}(i)) \}, \text{tl}(\text{tl}(i)))) \]
\[ > \]

Therefore \( \text{Proj}_{<3,(y)>}(T) = <(3, (\{ y \mapsto \text{hd}(\text{tl}(i)) \}, \text{tl}(\text{tl}(i))))> \).

Let \( P' \) be a slice of \( P \) constructed with respect to the slicing criterion \((3, \{ y \})\) according to definition 2.2.9, and let the state trajectory produced by the execution of \( P' \) in the initial state \((\sigma, i)\) be \( T' \). By definition 2.2.9, \( \text{Proj}_{<3,(y)>}(T) \) must be \( \text{Proj}_{<3,(y)>}(T') \), so \( \text{Proj}_{<3,(y)>}(T') \) will be \( <(3, (\{ y \mapsto \text{hd}(\text{tl}(i)) \}, \text{tl}(\text{tl}(i))))> \).

Clearly therefore, any valid slice, \( P' \) of \( P \) with respect to \((3, \{ y \})\) must affect the value of the variable \( y \). Since the only statement in \( P \) which does this is statement 2, statement 2 must be included in the slice. Furthermore, if the slice \( P' \) were to contain only statement 2, then \( \text{Proj}_{<3,(y)>}(T') \) would be \( <(3, (\{ y \mapsto \text{hd}(i) \}, \text{tl}(i)))> \). Therefore, in order to satisfy definition 2.2.9, statement 1 must also be included in \( P' \). The only valid slice of \( P \) w.r.t. \((3, \{ y \})\) is therefore \( P \) itself. However, this is not the slice produced by currently published static slicing algorithms [82, 52, 26], (all of which delete line 1).

Existing algorithms fail to produce the correct slice because, according to the standard definition of defined and referenced variables (definition 3.2.1),

\[ \text{DEF}[\text{scanf}("\%d", \&x);] = \{ x \} \]

and

\[ \text{REF}[\text{scanf}("\%d", \&x);] = \{ \} \]
This means that there will be no $du$-chain [5] between nodes 1 and 2 in the program $P$ in Figure 3.1. This is an example of a more general problem concerning the ‘implicit state’ [38, 39, 48].

To see why traditional formulations of defined and referenced variables do not cater for input statements correctly, we need to examine the state in more detail. It will be shown that by making the implicit state explicit the problem can be overcome.

**Definition 3.4.1 (The Explicit State)**

The *explicit state* is $(I \rightarrow V \cup seq(V))$, where $I$ is the set of variable identifiers and $V$ is the set of denotable values.

**Definition 3.4.2 (The Implicit State)**

The *implicit state* is any part of the state which is not explicit. That is, the implicit state consists of those state components which are not denoted by a variable identifier.

Observe that, whilst the effect of a program has upon the values stored in its variables is explicit, the effect it has upon the input sequence is implicit.

Existing slicing algorithms will include a statement $n$ in a slice iff:

1. the slicing criterion is transitively data dependent on $n$ or,
2. the slicing criterion is transitively control dependent on $n$.

Data dependence arises because of variable assignments (or, more generally, because of statements which affect the explicit state). Changes to the implicit state do not lead to dependencies as there is no variable to carry the dependence. In the most extreme case suppose a node $s$ (other than a predicate node), affects only the implicit state; No slicing criterion can be transitively control or data dependent upon $s$, and therefore, a slicing algorithm will be free to delete $s$.

Consider, for example, the program\(^2\) in Figure 3.2. Suppose the slicing criterion is $(3, \{y\})$. The slicing algorithm will be free to delete line 1 because the slicing criterion is neither tran-

---

\(^2\)Where `getint()` has the sole purpose of consuming an integer from the input.
satively control nor data dependent upon it. Indeed, line 1 may be deleted in the construction of any slice.

However, removal of line 1 clearly does affect the meaning of line 2. That is, in the original program, line 2 reads the second input into y, whereas, if line 1 is removed, it reads the first. Therefore, removing line 1 will produce a reduced program which does not preserve the effect of the original upon the final value of y. Such a reduced program is not a slice of the original according to Weiser’s definition of a slice (definition 2.2.9).

This discussion assumes that input statements read input from a single input device. For some programs it may be possible to view each input statement as reading data from a separate device, in which case existing slicing algorithms will produce correct slices.

3.5 Denoting the Input Sequence

The solution to the problem lies not in altering the slicing algorithm, rather it requires a change to the value of defined and referenced variable sets (upon which the algorithm depends). This is achieved by a reformulation of the implicit state as an explicit state [38, 39, 48], rather than altering the slicing algorithm, which constructs these slices.

Observe that, because

$$M_I[\text{scanf}("%d", \&x);](\sigma, i) = (\sigma \oplus \{x\} \leftarrow ld(i)), tl(i))$$

it will be inferred that
DEF[scanf("%d",&x);] = {x}

and

REF[scanf("%d",&x);] = {}.

because the only variable which alters its value in $\sigma$ is $x$ and this change references (depends upon) the value of no other mapping in $\sigma$. That is, although the input statement affects the implicit state, it does not affect the explicit state. Therefore the defined and referenced variable sets will not capture the linkage between successive input statements; this linkage consists of 'implicit du-chains'.

In order to remove the implicit state we need a new variable (and possibly a new denotable value [73] — the list), to denote the implicit state component. In this case, the pseudo-variable II shall be used to denote the input list.

Let $M_E$ describe the meaning of a statement in terms of the explicit state.

$$M_E[\text{scanf}("%d",\&x);]_{\sigma} =$$

$$\sigma \oplus \{ [x] \mapsto h d(\sigma[\text{II}]), [\text{II}] \mapsto t l(\sigma[\text{II}]) \}$$

from which it will be inferred that

$$DEF[\text{scanf}("%d",\&x);] = \{x, \text{II}\}$$

and

$$REF[\text{scanf}("%d",\&x);] = \{\text{II}\}.$$
The function $\Phi$, takes an implicit state and transforms into an equivalent explicit state:

$$\Phi : (I \rightarrow V) \times \text{seq}(V) \rightarrow (I \rightarrow V \cup \text{seq}(V))$$

$$\Phi(\sigma, i) = \sigma \oplus \{[\Pi] \leftarrow i\}$$

The connection between $M_I$ and $M_E$ is

$$\forall s. M_I(s) \circ \Phi = \Phi \circ M_E(s)$$

The relationship between $M_E$, $M_I$ and $\Phi$ is represented in the commutative diagram below:

$$
\begin{array}{ccc}
(I \rightarrow V) \times \text{seq}(V) & \xrightarrow{M_I(s)} & (I \rightarrow V) \times \text{seq}(V) \\
\downarrow \Phi & & \downarrow \Phi \\
(I \rightarrow V \cup \text{seq}(V)) & \xrightarrow{M_E(s)} & (I \rightarrow V \cup \text{seq}(V))
\end{array}
$$

Observe that this reformulation of the implicit state as an explicit state could have been achieved by re-writing the program, introducing assignments to the new pseudo-variable $\Pi$. The transformation, $T$, takes a statement $s$, and produces a statement $s'$, where $s'$ neither depends upon nor affects implicit state. The transformation $T$ for our simple while loop language is defined in Figure 3.3.

Observe that $(M_E(T(p)))(\Phi \sigma) = \Phi(M_I(p)\sigma)$, thus $T$ is guaranteed to remove the implicit state by source-to-source transformation. This could be established more formally by a simple
Figure 3.3: Removing the Implicit State from a Language with Input Statements

\[
\begin{align*}
T[\text{scanf}(s, &i_1, \ldots, &i_n);] &= I[i_1] \ldots I[i_n] \\
T[\text{while}(e);] &= \{\text{while}(e)\{T[e]\}\} \\
T[\{c_1 \ldots c_n\}] &= \{T[c_1] \ldots T[c_n]\} \\
T[\text{if}(e);] &= \{\text{if}(e)\{T[e]\}\} \\
T[\text{getint}();] &= \{II=tl(II);i\} \\
T[i=;] &= \{i=;\} \\
T[i] &= \{i = hd(II); II = tl(II);\}
\end{align*}
\]

Figure 3.4: A Simple Thermostat Program

```
1    reset();
2    while (inoperation()) { 
3      if (gettemp())
4        switchoff();
5      else switchon();
6      userchoice();
}
```

structural induction on the structure of the language.

### 3.6 A Thermostat Program

Often, in real-time systems, there will be a set of primitive commands for controlling input and output using devices such as sensors and actuators. These primitive commands will form part of a control language. Such programs may be hard to slice in any meaningful way, because we shall not be able to identify the interesting properties of the system in slicing criteria — they will all be implicit.

Consider, the (highly idealised) thermostat control program in Figure 3.4. As it stands this program is completely unslicable, as it mentions no variables.

If we model the implicit state using pseudo variables, we shall be able to transform programs such as the thermostat program into longer, but slicable, explicit versions. This corresponds to modelling the unavailable bodies of the primitive functions of the control language. In order to perform this transformation for the thermostat program we will need a specifica-
Figure 3.5: Informal Semantics for the Thermostat Language Primitives

<table>
<thead>
<tr>
<th>Pseudo Variable</th>
<th>Type</th>
<th>Description of Implicit State Modelled</th>
</tr>
</thead>
<tbody>
<tr>
<td>ideal</td>
<td>int</td>
<td>The ideal temperature</td>
</tr>
<tr>
<td>IsManual</td>
<td>boolean</td>
<td>True iff the system is in manual mode</td>
</tr>
<tr>
<td>temp</td>
<td>int</td>
<td>Current temperature reading on the thermometer</td>
</tr>
<tr>
<td>IsHeaterOff</td>
<td>boolean</td>
<td>True iff the heater is off</td>
</tr>
<tr>
<td>H</td>
<td>list(choice)</td>
<td>The User's list of inputs</td>
</tr>
</tbody>
</table>

Figure 3.6: Pseudo-Variables and the Implicit State Components they Denote

of the effect of each of the primitives. In this case, the device language primitives control and depend upon a thermometer and a heater. Figure 3.5 informally specifies the meaning of each primitive of the control language.

Figure 3.6 describes the implicit state value denoted by each pseudo variable we shall introduce.

Notice that we could, for all such problems, use a single variable to capture the entire implicit state [67]. This would require us to model the implicit state as a large data structure, denoted be a single variable. Using a single variable, primitive commands which depend upon or affect the implicit state would be transformed into commands which select and update parts of this data structure. Whilst this approach is theoretically acceptable, it is impractical, as it will dramatically reduce the precision of any slicing algorithm which depends upon it.

Figure 3.7 describes the transformation function for removing the implicit state. For the userchoice() primitive, the enumeration type \{up, down, manual\} is used to model the user's input. In this case, the input comes from a control device, which only offers three 'switch' buttons — up, down and manual. It will therefore, not be possible for the user to
The application of transformation rules from Figure 3.7 to the program in Figure 3.4 is depicted Figure 3.8.

We have adopted a decimal point numbering system to allow us to relate elements of the transformed program to those of the original (via their integral values).

Slicing with respect to \((6, \{ideal\})\) yields the slice depicted in Figure 3.9.

Converting this slice back to the original program notation we take the integral part of each statement as the members of the slice, thereby including a statement from the original if any of its transformed counterparts are in the slice of the explicit version.

<table>
<thead>
<tr>
<th>Transformation Rule</th>
<th>Original Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T[\text{while}(e)c] = [\text{while}(E[e]){T[c]}])</td>
<td>(T[\text{while}(e)c] = [\text{while}(E[e]){T[c]}])</td>
</tr>
<tr>
<td>(T[{e_1 \ldots e_n}] = [{T[e_1] \ldots T[e_n]}])</td>
<td>(T[{e_1 \ldots e_n}] = [{T[e_1] \ldots T[e_n]}])</td>
</tr>
<tr>
<td>(T[\text{if}(e)c] = [\text{if}(E[e]){T[c]}])</td>
<td>(T[\text{if}(e)c] = [\text{if}(E[e]){T[c]}])</td>
</tr>
<tr>
<td>(T[i:=c] = [i:=E[e]])</td>
<td>(T[i:=c] = [i:=E[e]])</td>
</tr>
<tr>
<td>(T[\text{reset}()] = [\text{IsManual} = \text{False}; \text{ideal} = \text{Default};])</td>
<td>(T[\text{reset}()] = [\text{IsManual} = \text{False}; \text{ideal} = \text{Default};])</td>
</tr>
<tr>
<td>(T[\text{switchoff}()] = [\text{IsHeaterOff} = \text{True}])</td>
<td>(T[\text{switchoff}()] = [\text{IsHeaterOff} = \text{False}])</td>
</tr>
<tr>
<td>(T[\text{switchon}()] = [\text{IsHeaterOff} = \text{False}])</td>
<td>(T[\text{switchon}()] = [\text{IsHeaterOff} = \text{False}])</td>
</tr>
<tr>
<td>(T[\text{userchoice}()] = [\text{if}(hd(\text{II}) == \text{up})\text{ideal} = \text{ideal} + 1;)</td>
<td>(T[\text{userchoice}()] = [\text{if}(hd(\text{II}) == \text{up})\text{ideal} = \text{ideal} + 1;)</td>
</tr>
<tr>
<td>(\text{else if}(hd(\text{II}) == \text{down})\text{ideal} = \text{ideal} - 1;)</td>
<td>(\text{else if}(hd(\text{II}) == \text{down})\text{ideal} = \text{ideal} - 1;)</td>
</tr>
<tr>
<td>(\text{else if}(hd(\text{II}) == \text{manual})\text{IsManual} = \text{True};)</td>
<td>(\text{else if}(hd(\text{II}) == \text{manual})\text{IsManual} = \text{True};)</td>
</tr>
<tr>
<td>(\text{II} = ti(\text{II});])</td>
<td>(\text{II} = ti(\text{II});])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transformation Rule</th>
<th>Original Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E[\text{inoperation}()] = [\text{II}])</td>
<td>(E[\text{inoperation}()] = [\text{II}])</td>
</tr>
<tr>
<td>(E[\text{gettemp}()] = [\text{read}(\text{temp});((\text{temp} &gt;= \text{ideal} - 2.0) &amp;&amp; (\text{temp} &lt;= \text{ideal} + 2.0))]])</td>
<td>(E[\text{gettemp}()] = [\text{read}(\text{temp});((\text{temp} &gt;= \text{ideal} - 2.0) &amp;&amp; (\text{temp} &lt;= \text{ideal} + 2.0))]])</td>
</tr>
<tr>
<td>(E[e_1 \ldots e_2] = E[e_1] \ldots E[e_2])</td>
<td>(E[e_1 \ldots e_2] = E[e_1] \ldots E[e_2])</td>
</tr>
</tbody>
</table>

**Figure 3.7:** Removing the Implicit State from the Thermostat Control Language
### 3.6 A Thermostat Program

<table>
<thead>
<tr>
<th>Original program</th>
<th>Transformed program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 IsManual = False;</td>
<td>1.1 IsManual = False;</td>
</tr>
<tr>
<td>1.2 ideal = Default;</td>
<td>1.2 ideal = Default;</td>
</tr>
<tr>
<td>2 while (!IsManual) {</td>
<td>2 while (!IsManual) {</td>
</tr>
<tr>
<td>3.1 read(temp);</td>
<td>3.1 read(temp);</td>
</tr>
<tr>
<td>3.2 if ((temp &gt;= ideal - 2.0) &amp; &amp; (temp &lt;= ideal + 2.0))</td>
<td>3.2 if ((temp &gt;= ideal - 2.0) &amp; &amp; (temp &lt;= ideal + 2.0))</td>
</tr>
<tr>
<td>4 switchoff();</td>
<td>4 IsHeaterOff = True;</td>
</tr>
<tr>
<td>5 else switchon();</td>
<td>5 else IsHeaterOff = False;</td>
</tr>
<tr>
<td>6 userchoice();</td>
<td>6.1 if (hd(II) == up)</td>
</tr>
<tr>
<td>6.2 ideal = ideal + 1;</td>
<td>6.2 ideal = ideal + 1;</td>
</tr>
<tr>
<td>6.3 else if (hd(II) == down)</td>
<td>6.3 else if (hd(II) == down)</td>
</tr>
<tr>
<td>6.4 ideal = ideal - 1;</td>
<td>6.4 ideal = ideal - 1;</td>
</tr>
<tr>
<td>6.5 else if (hd(II) == manual)</td>
<td>6.5 else if (hd(II) == manual)</td>
</tr>
<tr>
<td>6.6 IsManual = True;</td>
<td>6.6 IsManual = True;</td>
</tr>
<tr>
<td>6.7 II = ti(II); }</td>
<td>6.7 II = ti(II); }</td>
</tr>
</tbody>
</table>

Figure 3.8: Original and Transformed Thermostat Program

<table>
<thead>
<tr>
<th>Sliced Transformed Program</th>
<th>Corresponding Sliced Original Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice w.r.t. (6, {ideal})</td>
<td>Corresponding Sliced Original Program</td>
</tr>
</tbody>
</table>

Figure 3.9: Slicing the Thermostat Program Using its Explicit Counterpart
Chapter 4

Analysis of Dynamic Memory Access using Amorphous Slicing

4.1 Introduction

Problems associated with understanding, verifying and re-engineering the way in which a system allocates and releases dynamic memory present significant challenges to the software maintainer. Because the questions underlying these problems are undecidable, no system can provide a completely fail safe certification. For example, in checking for memory leaks, a system can only warn of potential problems, but cannot guarantee that no leaks remain.

We present an approach to modelling the dynamic memory access properties of a program using amorphous program slicing to create a Dynamic Memory Model (DMM). The slices are constructed from a transformed version of the original program in which heap access has been made explicit using a pseudo variable to denote the top of the heap. The DMM is a simplified version of the original program which is concerned solely with the dynamic memory access behaviour of the original. We illustrate the use of DMMs in problems of comprehension, verification and re-engineering.

We introduce a proof-of-concept DMM construction algorithm, showing how slicing simplification power can be dramatically improved using domain-specific transformation rules.
4.2 Overview

Most systems allocate and deallocate memory as they execute. This dynamic memory allocation is often hard to understand and problems associated with dynamic memory are notoriously difficult to model, predict and verify. For example, the cause of the London Ambulance Service System’s second crash at 2am on November 4, 1992 was blamed on the accidental insertion of a memory leak in the vehicle mobilisation routine [24] (section 4039).

In this chapter we introduce an approach modelling dynamic memory access properties using a mixture of implicit state removal [71] and amorphous slicing [44]. The focus of this work is an analysis of the quantity of dynamic memory allocated, rather than analysis of other dynamic memory access problems such as dangling pointers and inappropriate de-referencing.

To motivate our approach consider the program fragment in the left section of Figure 4.1, where \( N \) is a symbolic compile-time constant. The right section of the figure contains the version of the program with an additional variable \( hp \), which stores the current value of the heap top. Finally the amorphous slice constructed with respect to the final value of the pseudo variable \( hp \) would give:

\[
hp = N \times \text{sizeof} (\text{struct } L) ;
\]

We call this form of amorphous slice a Dynamic Memory Model (DMM). The DMM captures the computation in the original program concerned with dynamic memory access and removes other code which is irrelevant. For example, in this case, the DMM clearly indicates the amount of heap store used by the original program.

This chapter contains two related contributions to the problem of understanding, verifying and re-engineering the dynamic memory access properties of a program:

1. Extracting and understanding dynamic memory properties is reformulated as a program simplification problem.

This enables us to divide the problem into an automated phase (program simplification) and a non-automated phase (reasoning about the simplified program). This separation will be necessary in any complete approach, as the underlying questions concerning dynamic memory consumption are undecidable.
The philosophy of ‘programs as approximate answers to undecidable analysis questions’ is an underlying theme of this research.

2. We show that the specific approach to simplification we advocate makes a set of powerful simplifying transformations applicable.

This is important because defining suitable transformation strategies remains an open problem. In this case we have a simple example of ‘domain specific program transformation’, where the domain knowledge is used to guide the transformation strategy. When used in a more generic transformation scenario, these domain specific transformations would be unlikely to yield such dramatic simplification.

The rest of this chapter is organised as follows: Section 4.3 defines an algorithm that makes the heap pointer explicit and section 4.4 briefly reviews amorphous program slicing, arguing that it is better suited to this problem than conventional slicing alone. Section 4.5 introduces some transformation rules for amorphous slicing of DMMs, showing how domain knowledge can be exploited to greatly improve simplification power. Section 4.6 presents a simple algorithm for computing amorphous DMM slices, which is illustrated with a worked example in section 4.7. Section 4.8 shows how DMMs can be applied to comprehension, analysis and re-engineering problems.
4.3 Modelling the Implicit State

Our first step is to make the dynamic memory properties of the program 'sliceable'. This is not possible with the original code, because there is no variable which denotes the amount of heap allocated. More precisely, there is no explicit state component which captures this - there is, of course, an implicit top of heap pointer which does so.

In chapter 3 used the term 'implicit state' for components of the state which do not form a part of the store. We view a store as a mapping from names to values, thus it has the properties of an environment in a functional language [73]. We use the term 'explicit state' to refer to this store, because its components have an explicit denotation (the variables) in the program and can thus form the subject of a slicing criterion. Chapter 3 also introduced a simple technique for denoting implicit state components with pseudo variables, making these components explicit. This transformation of an implicit-state program into an explicit-state program is useful because it allows us to form a slicing criterion which captures the (previously implicit) state. In this chapter we use the implicit-state removal approach from chapter 3 to make slicing applicable to Dynamic Memory Modelling.

The implicit state removal algorithm for a simple intraprocedural language (involving heap allocation using malloc) is presented as a context-free, syntax directed transformation in Figure 4.2. The effect of this simple linear-time source-to-source transformation is to insert assignments into the program to a new variable, hp, which ensure that hp stores the current amount of heap store allocated. Thus all programs start with the following initial assignment

\[ hp = 0; \]

Where ever there is an allocation of memory in the original program, the transformed version will contain a re-assignment to hp.

We have made some simplifying assumptions about the kind of program to which this transformation will be applied. For example, we assume that calls to malloc can only occur as the right hand sides of assignments and that the actual parameter to malloc does not, itself, contain a call to malloc. These merely serve to simplify the exposition; a more complex implicit-state removal transformation can be defined to handle the more general case where
malloc calls can occur in arbitrary expressions.

Also, we assume that expressions are side effect free.

### 4.4 Amorphous Slicing

Having introduced the pseudo variable hp to capture the implicit computation on the top of the heap, we are left with a program which is longer than that with which we started. However, the explicit version of the program is amenable to slicing.

A conventional program slice [82, 52, 35] is an abstraction of a program which contains the computation associated with some set of variables at some point of interest. The pair \((V, n)\), containing the set of variables of interest, \(V\) and the point of interest, \(n\) is known as the slicing criterion. The process of slicing consists of identifying all statements and predicates which can have no effect upon the value of any variable in \(V\) when the next statement to
be executed is at point \( n \). A slice is thus both a syntactic and a semantic projection of the program from which it is constructed. Detailed surveys of program slicing can be found in papers by Tip [76] and Binkley and Gallagher [15]. There is also an excellent ‘slicing home page’ maintained by Jens Krinke [57].

Amorphous slicing [41] is a variation of conventional slicing in which the slice must preserve the effect of the original program with respect to the slicing criterion, whilst it need not be a syntactic subset of the original. Amorphous slicing therefore retains the semantic aspect of conventional slicing, but removes the syntactic restriction to command deletion.

In this chapter, we shall only require amorphous slices to be constructed for the single variable \( hp \) at the end of the program, so the amorphous slicing criterion will always be \((\{hp\}, L)\), where \( L \) is the end node of the program’s CFG.

The relationship between amorphous and conventional slicing is discussed in more detail in [41]. Here, we simply note that amorphous slices are never thicker and often considerably thinner than corresponding conventional slices. The extra simplicity is achieved because the amorphous slice need not syntactically resemble the original program. This makes the approach inappropriate for some applications of slicing (for example cohesion measurement [65, 11], and debugging [62]).

However, slicing has also been applied to comprehension [28, 54] and reuse [20, 23] and for these applications amorphous slicing is more suitable, as slice simplicity is more important than syntactic preservation. Indeed, in cases where we have previously applied an implicit state removal transformation, we will have already deviated from the syntax of the original source code, and so there is nothing to be lost by applying amorphous slicing in place of conventional slicing.

To construct a DMM of a program \( p \), we use amorphous slicing at the end of \( T[p] \) with respect to the variable \( hp \). This produces a program which is as simple as possible but which has the same effect as \( T[p] \) on the variable \( hp \). Thus, the DMM captures in \( hp \) the effect of \( p \) on the top of the heap.

**Definition 4.4.1** (DMM)
Let $p$ be a program or program fragment. The DMM of $p$ is the amorphous slice constructed for the program $T[p]$ with respect to the slicing criterion $(\{hp\}, L)$, where $L$ is the end node of the CFG of $T[p]$.

### 4.5 Domain Specific Transformation

Transformation rules suitable for creating DMMs from our simple language are listed in Figure 4.3. The rules are written in the form of a quasi logical calculus, so they are not fixed to the transformation strategy we use in this chapter (presented in section 4.6).

A rule of the form: $\frac{A}{B \rightarrow C}$ can be interpreted as "If $A$ holds then the fragment $B$ can be transformed to the fragment $C$".

The terms $\text{REF}(e)$ and $\text{DEF}(e)$ denote the referenced and defined variables of $e$ (we employ a slight abuse of notation by assuming that these two functions may be applied to either expressions or to commands). The term $\text{SUB}(e_1, i, e_2)$ returns the expression that results from substituting all occurrences of the variable $i$ in the expression $e_1$, with the expression $e_2$.

Three of these transformations (‘Collapse Incremental For Loop’, ‘Collapse Decremental For Loop’ and ‘Collapse If-Else’) are worthy of special note. They resemble strength reducing operations on expressions used in peep-hole compiler optimisation [5] and in partial evaluation [17]. The collapsing transformations reduce the strength of statements, by identifying situations in which non-primitive statements (for or if) can be replaced by a simple assignment.

Because of their simplifying power, these collapsing rules would be very useful in program analysis where they to be widely applicable. However, in most ‘normal’ programs they will be far from widely applicable and so a general transformation strategy would be unlikely to make progress using them.

However, the programs we shall be constructing amorphous slices from are far from normal. We claim that it is not unlikely that a programmer will write (perfectly sensible) code that will make these two rules applicable when applied to the problem of constructing DMMs.
Conditionals

It is not unlikely that the programmer will write code which contains a conditional in which the same amount of memory is allocated in both the *then* and *else* branches (even though this memory may be put to entirely different uses). In such cases the Collapse If-Else axiom will be applicable, once the implicit state removal transformation, $T$, has been applied.

Loops

It is not unlikely that a programmer will write a loop to create some recursive data structure (such as a list or tree). At each execution of the loop a constant amount of memory will be consumed. If the loop executes for a bounded number of iterations (as is the case with many *for* loops) and this bound can be inferred as a compile-time expression, then transformations like the *for* loop collapsing rules will become applicable.

### 4.6 A Simple Algorithm for Intraprocedural Amorphous DMM Slicing

The author has implemented a prototype tool, based upon a simple algorithm which produces DMMs from single procedure programs. In its current form the tool merely serves to demonstrate ‘proof of concept’, as it handles neither the interprocedural case, nor aliasing.

Nonetheless, this simple algorithm is capable of producing the amorphous slices presented in this chapter, and we believe that it demonstrates the feasibility of and justification for the concept of a Dynamic Memory Model.

The algorithm is presented in Figure 4.4, a larger example is presented in Figure 4.6 and two ancillary transformation tactics are described in Figures 4.7 and 4.8. The essential idea is to use the push and pull rules to expand expressions and to coalesce assignments to the same variable, in a manner reminiscent of symbolic execution strategies. This has the effect of reducing the interdependencies between statements at the expense of enlarging
4.6 A Simple Algorithm for Intraprocedural Amorphous DMM Slicing

Figure 4.3: Transformation Rules for Amorphous DMM Slices
Step 1: Introduce a variable $hp$ using the transformation, $T$, in Figure 4.2
Step 2: Construct a conventional slice for the criterion $(\{hp\}, L)$
Step 3.1: Use SE to symbolically execute (see Figure 4.7)
Step 3.2: Construct the conventional slice for the criterion $(\{hp\}, L)$
Step 3.3: Apply the collapsing rules and while to for where possible
Step 3.4: If any collapsing rule was applicable then repeat from Step 3.1

Figure 4.4: The Top Level algorithm

their expressions. The reduction in inter-dependencies increases the deletion opportunities available to conventional slicing.

Having simplified the program by expanding expressions and slicing away the unwanted statements, the collapsing rules and axioms may become applicable. If they are, then they will introduce additional assignment statements (in place of conditionals and loops) and these new assignments can be pushed using the ‘symbolic execution’ phase. These ‘pushes’ in turn may make more slicing and collapsing possible. The algorithm iterates until no further collapsing rules apply.

Termination of the algorithm is guaranteed because the collapsing rules guarantee to reduce program size and if none apply then the algorithm halts. The algorithm essentially has quadratic complexity because it involves a pass of (potentially) the whole program for each assignment statement. This complexity makes the algorithm of value only for intraprocedural amorphous slicing, as to perform interprocedural amorphous slicing on large programs would be prohibitively expensive. Fortunately, work is in progress [13] on an implementation of amorphous slicing using transformations on program dependence graphs, which will produce a more realistic interprocedural algorithm.

4.7 Worked Example

Consider the “exam marks” program in Figure 4.5. The result of removing the implicit state (Step 1) is given in the left section of the Figure 4.6 and the result of constructing an initial conventional slice (Step 2) is given in the right section of the Figure 4.6.

The conventional slice is the starting point for the iterative steps of the algorithm. It
pass = NULL;
fail = NULL;
passnum=0;

printf("Number of students?\n");
scanf("%d",&num);

for (i=0;i<num;i=i+1) {
  e = (exam) malloc(32);
  printf("\n%d name: ",i+1);
  scanf("%s",&name);
  printf("\n%d mark: ",i+1);
  scanf("%d",&marks);
  strcpy(e->name,name);
  e->marks = marks;
  r = (list) malloc(10);
  r->datum = e;

  if (marks >= 40) {
    r->next = pass;
    pass = r;
    passnum = passnum+1;
  } else {
    r->next = fail;
    fail = r;
  }
}

printf("\n\n");
for (i=0;i<passnum;i=i+1) {
  e = pass->datum;
  printf("\nName: %s",e->name);
  printf("\nMark: %d",e->marks);
  pass = pass->next;
}

Figure 4.5: Exam Marks Original Program
hp = 0;
pass = NULL;
fail = NULL;
passnum=0;
printf("Number of students?");
scanf("%d",&num);

for (i=0;i<num;i=i+1) {
    hp = hp + 32;
e = (exam) malloc(32);
printf("\n%d name: ",i+1);
scanf("%s",&name);
printf("\n%d mark: ",i+1);
scanf("%d",&marks);
strcpy(e->name,name);
e->marks = marks;
hp = hp + 10;
r = (list) malloc(10);
r->datum = e;
if (marks >= 40) {
    r->next = pass;
pass = r;
passnum = passnum+1;
} else {
    r->next = fail;
fail = r;
}
}
printf("\n\n");
for (i=0;i<passnum;i=i+1) {
e = pass->datum;
printf("\nName: %s",e->name);
printf("\nMark: %d",e->marks);
pass = pass->next;
}

Implicit State Removed

Conventional Slice

Figure 4.6: Exam Marks Program
4.7 Worked Example

**Figure 4.7: Symbolic Execution Algorithm**

\[
\text{SE} : \text{listof}(C) \rightarrow \text{listof}(C)
\]

SE(sl) is defined as follows

\[
\begin{align*}
\text{set } & \text{ WorkList := sl, DoneList := } \emptyset, \text{ TopList := } \emptyset \\
\text{while } \text{ WorkList not empty } & \text{ do} \\
\text{if } \text{ head(WorkList) is a for, if or while } & \text{ then} \\
\quad \text{Apply SE to the body of head(WorkList)} \\
\text{else} \\
\quad \text{if } \text{ head(WorkList) is an assignment } & \text{ then} \\
\quad\quad \text{set } (c, cl, stuck) := \\
\quad\quad \quad \text{PUSH(head(WorkList),tail(WorkList))} \\
\quad\quad \text{if not(stuck) } & \text{ then} \\
\quad\quad\quad \text{set } \text{ WorkList := cl} \\
\quad\quad\quad \text{Append } c \text{ to DoneList} \\
\quad\quad\text{else } \text{ Append } c \text{ to TopList fi} \\
\quad\text{else } & \text{ Append } c \text{ to TopList fi} \\
\text{fi} \\
\text{set } \text{ WorkList := tail(WorkList) od} \\
\text{return } \text{ Append(TopList, DoneList)}
\end{align*}
\]

is not required for correctness, but as this example demonstrates, conventionally slicing the program does tend to remove a lot of code which serves to reduce the effort required by the transformation/slicing phase which follows it.

Executing Step 3.1 has the effect of collecting together the two assignments to \(hp\) inside the `for` loop (using the Unfold Assign rule). This transforms\(^1\) the DMM to:

\[
\begin{align*}
\text{hp} & = 0; \\
\text{scanf("%d",&num) ;} \\
\text{for}(i=0; \text{i<num}; \text{i=i+1}) \\
\quad \text{hp} & = \text{hp+42;}
\end{align*}
\]

Step 3.2 (conventional slicing) has no effect upon this transformed program, but Step 3.3 (the Collapsing stage) reduces it to:

\[
\begin{align*}
\text{hp} & = 0;
\end{align*}
\]

\(^1\)Our prototype tool does reduce constant expressions, though it currently performs no other expression level simplification.
PUSH : $C \times listof(C) \rightarrow C \times listof(C) \times bool$

PUSH($c, cl$) is defined as follows

set WorkList := $cl$, DoneList := $\emptyset$

while WorkList not empty do

set $[i = c] := c$

if head(WorkList) is an assignment then

set $[i' = c'] :=$ head(WorkList)

if $i = i'$ then

return ($\emptyset$, Append(DoneList, WorkList), $true$)

else

if Push Assignment rule applies then

set $[c_1; c] :=$ result of Push Assignment rule

Append $c_1$ onto the end of DoneList

else

return ($c$, Append(DoneList, WorkList), $true$)

fi

else

if Push If or Push If-Else rules apply then

set $[p; c] :=$ result of Push rule

Append $p$ onto the end of DoneList

else

return ($c$, Append(DoneList, WorkList), $true$)

else

return ($c$, Append(DoneList, WorkList), $true$)

fi

set WorkList := tail(WorkList)

od

return ($c$, DoneList, $false$)

Figure 4.8: Push Algorithm
4.7 Worked Example

\begin{verbatim}
scanf("%d",&num) ;  
hp = hp+(num-0)*42;
\end{verbatim}

Since some collapsing took place, Step 3.4 returns us to Step 3.1, which produces:

\begin{verbatim}
scanf("%d",&num) ;  
hp = 0+(num-0)*42;
\end{verbatim}

Conventional slicing (Step 3.2) and the collapsing rules (Step 3.3) have no effect, so the algorithm terminates with step 3.4.

It is interesting to note that although this DMM is attractively small, its very simplicity could be regarded as a disadvantage; for example, we have lost the \texttt{printf} statement which accompanied the \texttt{scanf} statement and which to some extent played the role of a comment in the way it described the value read in by the \texttt{scanf}. This unfortunate ‘loss of context’ is a property of all slicing-based approaches to simplification. It is a property which we could ameliorate using origin tracking [31]. This is an approach used by the PIM system [30], which constructs slices by term rewriting. Each syntactic term maintains a link to the context (its origin) from which it came prior to the application of the rewrite rule which produced it.

Using the algorithm in Figure 4.4, we have thus constructed from the original program a small, simple model of its behaviour with regard to the heap store.

At this point it is only fair to make clear that the dramatic simplification that we have been able to achieve is possible because the programs we have considered contained loops for which the number of iterations could be determined as a compile-time expression. However, while this may not always be the case, we shall clearly be able to exploit this property in any fragments where it does apply. Also, as might be expected, we have found that many of the programs we have examined contain a large amount of code which does not contribute to the heap access characteristics of the overall program. This guarantees that slicing will be an attractive technique for focusing upon those parts of the program which do involve the heap.
4.8 The Use of DMMs in Analysis

In this section we consider a few applications to which DMMs may be put. The list is not exhaustive, as a DMM is simply a semantic abstraction of a program’s heap computation and therefore can be used in any task which requires analysis of this behaviour.

4.8.1 Re-Engineering

In some cases it may be possible to use the DMM to identify re-engineering opportunities. It is likely that some sections of code will allocate an amount of dynamic memory which is bounded and for which the upper bound can be deduced at compile time. From the original program, it will not be immediately obvious which sections of code have this property and for those which do, it will not be obvious what the upper bound is. Similarly, we might like to deduce a lower bound for memory allocation.

Using the DMM, the bounds for any allocation of dynamic memory will be easier to deduce, because all computation not involving memory allocation will be stripped away and that which remains will ‘homogenise’ all memory allocations as simple integer operations on \( h \). As all slicing algorithms are conservative, we cannot guarantee that all such ‘irrelevant’ computation will be removed. However, our algorithm is safe; if code is not present in the DMM then such code is definitely irrelevant.

4.8.2 Space Complexity Formulae

In addition to extracting lower and upper bounds on memory allocation, it will also be useful to construct a formula which expresses the amount of store used by a subsystem in terms of some determining parameters. This is a hard problem for a maintainer presented with an entire system, but the DMM presents this information in isolation and in the familiar syntax of the original programming language.

For example, suppose a large system contains a fragment of code like that depicted in Figure 4.9. This fragment creates a table, allocating row elements with \texttt{AllocateRow} and column elements with \texttt{AllocateCol}, each of which takes a pointer and creates enough space
4.8 The Use of DMMs in Analysis

```
hp = 0 ;
for(i=0;i<n;i=i+1)
   { AllocateRow(NewRow) ;
     Row = AppendRow(Row,RowCalc(NewRow,i)) ;
     for(j=0;j<m;j=j+1)
        { AllocateCol(NewCol) ;
          Col = AppendCol(Col,ColCalc(NewCol,i,j))
        }
   }
```

```
hp = (n-0)*((r+(m-0)*c)) ;
```

<table>
<thead>
<tr>
<th>Original Program</th>
<th>Dynamic Memory Model</th>
</tr>
</thead>
</table>

Figure 4.9: Using DMMs to Capture Space Complexity of Underlying Algorithms

for the structure it references. The functions RowCalc and ColCalc establish the values of row and column elements in terms of the current row number and in terms of the current row and column number respectively.

Suppose the amount of store consumed by a call to AllocateRow is \( r \) cells and that consumed by AllocateCol is \( c \) cells. Observe that rows are allocated in the outer loop (which executes \( n \) times), whereas column elements are allocated in the inner loop (which executes \( nm \) times). We could reasonably expect that the fragment will consume \( rn + cnm \) store cells in total.

The maintainers could attempt to establish this for themselves by manual inspection. However, this would be a time-consuming and costly activity. On the other hand, using our DMM construction algorithm, we obtain the DMM shown in the rightmost section of Figure 4.9, revealing at a glance the space complexity of the original code fragment.

We could simplify this DMM further by applying simple algebraic identities to simplify expressions. We have not yet written such expression-level simplification rules into our prototype tool. This form of expression simplification is included in comparable systems such as PIM [30] and the maintainers’ assistant [10, 22], and we plan to augment our tool in a similar fashion, to improve the quality of the DMMs it produces.
Consider the program fragment in the leftmost section of Figure 4.10. In the original fragment \texttt{getmem()} is a function which allocates \texttt{const} bytes of store, while \texttt{releasemem(s)} returns \texttt{const} bytes of store to the free list starting at address \texttt{s}. The part of the fragment marked with three vertical dots indicates an arbitrary section of code, which neither allocates nor releases dynamic memory.

The central section of the figure contains the result of removing the implicit state and the final section contains the DMM. Once again the ability to simplify expressions would improve the algorithm; we would be able to transform the assignment

\[
\text{hp} = (\text{const} + (N-1) \times \text{const}) - (N-1) \times \text{const};
\]

to produce the final DMM:

\[
\text{hp} = \text{const};
\]
... demonstrating the presence of a memory leak of \texttt{const} bytes even more starkly.

Of course this simple program only allocated memory. We have not covered dynamic memory deallocation in the implicit state removal transformation, $T$, presented in Figure 4.2. To do so is not hard, but it does require the retention of a symbol table, which complicates the exposition somewhat. More work is required to define implicit state removal for more elaborate memory management systems and to explore the transformation sets which are most appropriate to the models which arise.
Chapter 5

Slice-Based Dynamic Memory Modelling — Case Studies

5.1 Introduction

Chapter 4 suggested that slicing can be used to extract the parts of a program which are relevant to the allocation and de-allocation of dynamic memory. In this way, the programmer can analyse the effect of the original program upon the system’s dynamic memory. Using the slice, the programmer can also assess the algorithmic space complexity of memory allocated and check for the presence of possible memory leaks. This chapter explores this claim, using 3 case studies of a real program: diff, the GNU differencing program, Word count and DVD stock information.

In order to construct our studies, we faced the well-studied problem of measuring the size of a program (in order to compare the performance of amorphous and syntax-preserving slicing). We chose to adopt the simple approach of counting Lines of Code (LoC). However, even this simple metric is the subject of much controversy [29, 69] and we found it necessary to perform a number of transformations to the original program and its slices, in order to ensure that sizes of slices were compared on a ‘level playing field’, using LoC.
The results of our case studies show:

- Syntax-preserving static slicing is applicable to the problem of dynamic memory analysis;

- Amorphous static slicing further improves the results of syntax-preserving static slicing;

- Syntax-preserving slicing performs surprisingly well;

- In one of the 3 studies, it turns out that the Conditioned slicing (in both syntax-preserving and amorphous forms) is highly desirable for dynamic memory modelling.

These results are interesting because syntax-preserving static slicing has now reached a maturity level, where existing tools such as the GrammaTech slicer [36] are publicly available and are capable of producing slices of the full range of C programming language features in reasonable time. Adapting such tools to the dynamic memory problem is therefore attractive.

Conditioned and amorphous slicing technology is comparatively new and less mature than its syntax-preserving counterpart [19, 25, 47]. However, the (semi-automatically produced) conditioned amorphous slices we were able to construct in diff program case study suggest that the combination of amorphous and conditioned slicing will be a very useful tool for dynamic memory analysis.

The rest of this chapter is organised as follows: Section 5.2 presents background material on slicing and the use of slicing to construct a dynamic memory model. Section 5.3 describes the approach taken to making the results from various slicing algorithms and approaches comparable using ‘Lines of Code’ (LoC) as a metric. This required some pre-processing to ensure that lines of code were counted in a fair and consistent manner and some post-processing to repair the (only slightly) erroneous slices produced by Unravel. Section 5.4 presents the results for static slicing, comparing syntax-preserving slicing, with amorphous slicing. The results are surprisingly good for syntax-preserving slicing and lead to Section 5.5 which considers the way in which the results for both syntax-preserving and amorphous slicing can be improved using conditioned slicing.
5.2 Dynamic Memory Modelling

A Dynamic Memory Model (DMM), is a slice of a program which captures the dynamic memory allocation and de-allocation behaviour of the original. In order to allow slicing to be applied, a pseudo variable, $hp$, is introduced to model the top of the heap. This section briefly reviews the DMM approach.

5.2.1 Syntax-Preserving Slicing

The method for reducing the number of statements, which still produce the same behaviour as the original program on specified variable(s), is called program slicing [82, 75, 27, 46]. The reduced program, called a slice, has the advantage of being executable, yet smaller than the original. The reduction in size is achieved by removing statements, which cannot affect the values of variables of interest. The variables (paired with the point in the program at which they are of interest) are known as the ‘slicing criterion’. In this chapter, we shall only be concerned with slicing at the end of the program, for a single variable so, for this chapter, the slicing criterion will consist of a single variable. Slicing deletes parts of the program, which cannot affect the final value of this variable.

For example, consider the simple code fragment in column (a) of Figure 5.1. Suppose that the slicing criterion is $x$. Slicing deletes unwanted statements from the program (in this case, those that have no effect on $x$). In Figure 5.1, the variable $x$ does not depend on $y$, hence the slice does not contain the second line. It also does not depend upon $r$ and so the final line is also deleted in forming the slice.

The slice of the program is shown in column (b) of the figure. This slice is called ‘syntax-
preserving’ [41, 79, 27], to distinguish it from an amorphous slice (which does not necessarily preserve syntax). The slice is also called a static slice, to distinguish it from a dynamic slice [56, 4] and a conditioned slice [28, 19]. In dynamic slicing the input to the program is fully known at slice time, because the program has been executed (whence ‘dynamic’). In conditioning slicing, the program has not been executed, but some information about the way in which the program will be executed is available, in the form of a condition on the initial state (whence ‘conditioned’).

Normally, sliced programs are thinner than the original program, but there are some occasions when the original program itself could be the sliced program. For example, had the slicing criterion been \( r \), then the slice would have been the whole program in the example of Figure 5.1.

### 5.2.2 The Pseudo Variable \( \text{hp} \)

The heap is an implicit part of the state of computation. That is, there is no variable in the program, which explicitly captures (in the syntax of the program) the final value of the heap.

In [71] a simple technique for denoting implicit state components with pseudo variables is introduced to make these components explicit. The approach, its correctness and semantic foundations are described in more detail in [71].

In the case of modelling the heap, the effect of this simple linear-time source-to-source transformation is to insert assignments into the program to a new variable, \( \text{hp} \), which ensure that \( \text{hp} \) stores the current amount of heap store allocated. Thus, all programs start with an initial assignment of \( \text{hp} = 0; \), and wherever there is an allocation of memory in the original program, the transformed version will contain a re-assignment to \( \text{hp} \).

Having performed this pre-transformation, the program’s dynamic memory access properties can be analyses by slicing upon the final value of the pseudo variable \( \text{hp} \).

Consider the program fragment in column (a) of Figure 5.2. Column (b) of the figure contains the version of the program with an additional pseudo variable \( \text{hp} \), which stores the current value of the heap top. Finally, column (c) contains the amorphous slice constructed with respect to the final value of the pseudo variable \( \text{hp} \).
This form of slice is called a Dynamic Memory Model (DMM) [49]. The DMM captures the computation in the original program concerned with dynamic memory access and removes other code which is irrelevant. For example, in this case, the DMM clearly indicates the amount of heap store used by the original program.

The DMM could have been constructed using syntax-preserving slicing. This would have produced a larger slice, but one which still captures the effect of the original upon the heap. The syntax-preserving slice would not have contained assignments to rec and n, but would have retained the loop and all assignments to i and hp. For dynamic memory modelling, the original program’s syntax is modified by the inclusion of the pseudo variable, hp. Therefore, since the goal is to construct a reduced program, there appears to be little reason to use syntax-preserving, rather than amorphous, slicing.

5.3 Preparation for Case Studies

In order to investigate the relative merits of various forms of slicing upon the size of the DMM, 3 case studies were constructed. The largest program chosen was \texttt{diff}. This was chosen because this program contains readily available course code, it performs dynamic memory allocation and consists of approximately 500 lines of source code. This makes it sufficiently small to analyse using existing amorphous slicing technology, yet sufficiently large.
to be a worthwhile and ‘practical’ example. All the source code and results are shown in Appendix A.

Syntax-preserving slices were constructed using the Unravel slicing tool [61], while amorphous slices were constructed using the amorphous slicing algorithm defined in [49].

The Word count and DVD stock analysis are relatively very small. Our investigation shows that the Word count program has no memory leaks and DVD stocking analysis program never frees any allocated memory. The source code and results of various slices are shown in Appendix B and Appendix C.

5.3.1 Assumptions in Counting Lines of Code

In order to measure the size of the slices produced by syntax-preserving and amorphous slicing, it was necessary to arrive at a ‘fair’ way of counting lines of code. We found that the diff program used a number of programming features which tended artificially to reduce our initial, naïve, measure of lines of code, thereby potentially biasing the results.

In addition to these coding conventions (which tended to reduce the LoC metric) there are also a number of lines of ‘code’, which are not statements (which tended to increase the LoC metric). These were not counted.

In order to ensure that, wherever possible, the same programming task was measured as containing the same number of lines of code, a number of pre-transformations were performed to rewrite the program in a canonical form. These re-write transformations are described in this section as a set of rules. Having applied the rewrite rules, it is safer to apply our naïve LoC metric for assessing the size of the original program and its various slices. The rest of this section defines these rules.

Rule 5.3.1 (Variable declarations)

All (non-initialising) variable declarations are ignored.

Eg: `int i;`

Variable declarations are not executable code. They are not contained in slices produced by Unravel, and it seems unfair to count them as lines of code.
However, the C language allows for declarations to include initialisation. The initialisation is an assignment, and as such, should count as a line of code. To clarify this, we observed that all such ‘initialising declarations’ could be re-written as pure (initialisation-free) declarations followed by assignments. Therefore we performed this transformation before counting lines of code.

Rule 5.3.2 (Special variable declarations)
When a variable is declared and initialised at the same instance then the transformation (exemplified by Figure 5.3) will be applied. After transforming, the assignment statement, \texttt{int length = 0;}, is thereby counted as contributing one line to the LoC metric.

Unravel does not take account of \texttt{#include} directives, and so these are ignored.

Rule 5.3.3 (#include)
\texttt{#include} preprocessor directives are deleted.

Compiler directives are not recognised by Unravel, and do not contribute to the computation of the DMM, if their bodies are code-free.

Rule 5.3.4 (#ifndef ... #endif)
Code-free compiler directive statements are ignored.

For example, \texttt{diff} contains the code:

\texttt{#ifndef GUTTER\_WIDTH\_MINIMUM}
This code is not counted as part of the LoC metric. However, there are compiler directives in `diff`, which call user-defined functions. These are counted, by treating the enclosing `#ifdef` as being `true`.

**Rule 5.3.5 (Special `#ifndef ... #endif`)**

Within the compiler directive statements, if it has a calling procedure or an ordinary statement in the body, then the statement will be taken into account.

For example, `diff` contains the directive:

```c
#ifdef SIGCHLD
    signal (SIGCHLD, SIG_DFL);
#endif
```

This is included, by rewriting it to:

```c
signal (SIGCHLD, SIG_DFL);
```

Which is counted as a single line for the purpose of computing the LoC metric.

Compound statements in C consist of a number of statements encased in open closing curly brackets. The curly brackets are not counted in forming a value of LoC.

**Rule 5.3.6 (Compound statement)**

The open '{' is placed at the start the first compound statement and similarly the close '}' is placed at the end of the last statement of the compound statement.

The `diff` program contains a number of instances of the conditional expression operator. This allows code, which might otherwise be written as a conditional statement to be rendered in terms of a single assignment involving a conditional expression. This tends to create compact code, and 'artificially' reduces the LoC metric. An example of the application of this rule to code from `diff` is given in Figure 5.4 and Figure 5.5.
Rule 5.3.7 (Conditional expression)

The conditional expression is re-written as `if` `then` `else` statement so that the semantics of the code fragment will be preserved.

```
f.desc=(strcmp(oarg,"-")==0 ? STDIN_FILENO :
  open(oarg,0,ONLY,0));
```

**Original**

![Figure 5.4: Conditional Expression](image)

```
if (strcmp(oarg,"-")==0) f.desc=STDIN_FILENO;
else f.desc=open(oarg,0,ONLY,0);
```

**After Rewriting**

![Figure 5.5: Conditional Statement Rewrite](image)

Boolean operators in C are evaluated using short-circuit evaluation. This effectively introduces sequence points [1], which should be taken into account in computing the size of a program. Clearly, a nested `if` `then` of arbitrary depth with a single assignment within the most deeply nested conditional, can be converted to a single `if` `then`, in which the conditions are conjoined using the `&` operator. To make the consideration of complex boolean operators consistent with corresponding expanded statements, the two short-circuit boolean operators `&` and `|` are re-written as conditional expressions. These will subsequently be re-written (using Rule 5.3.7), to nested conditional statements.

Rule 5.3.8 (And operator)

```
p & q => p?q:0
```

The `&` operator can be re-written as a conditional operator.

Rule 5.3.9 (Or operator)
Similar to Rule 5.3.8, the ‘||’ operator can be re-written as a conditional operator.

Example 5.3.1 (Rewriting Example)

The single statement\(^1\) shown in Figure 5.6 is used in \texttt{diff}. This statement needs to be simplified, in order fairly and consistently to count the LoC metric. The code fragment is transformed by using the ‘Conditional’, ‘And’ and ‘Or’ rules (that is, rules 5.3.7, 5.3.8, and 5.3.9) to the result in Figure 5.7.

This rewriting has the effect of expanding out compact code, which attracts a low count for LoC, to a more basic form, which attracts a higher score for LoC. The rewritten code is ‘more basic’ because conditional expressions and boolean operators are not used and preprocessor directives are deleted or instantiated.

\begin{verbatim}
1 no_diff_means_no_output=
 (output_style==OUTPUT_IFDEF?
  (!*group_format[UNCHANGED] ||
   (strcmp(group_format[UNCHANGED],"%=")==0 &&
    !*line_format[UNCHANGED]))):
 output_style ==
 OUTPUT_SDIFF?sdiff.skip_common_lines:1);
\end{verbatim}

Figure 5.6: An original complex single statement from \texttt{diff}

\(^1\)Since long lines cannot be displayed on a single line within the chapter, line numbers are used to indicate the LoC count which a piece of code will attract.
5.3.2 Modifying the output of Unravel

Slicing unstructured programs is known to present problems [2, 6, 21, 42]. Unravel suffers from these problems; it does not produce a correct slice in the presence of jump statements. Rather, it simply ignores all such statements. Although diff is a structured program, and does not contain any goto statements, like almost all C programs, it does contain a large number of break statements. These are used to make a switch statement into a more traditional case-style statement, in which only one arm of the case statement is executed when any case is selected for execution.

Unfortunately, Unravel treats break statements as jumps and, therefore, does not include them in any slices.

We also found that Unravel does not include one of the functions (add_exclude) which it should have done, and so we added this function to the result produced by Unravel to form the syntax-preserving slice. Thus, the results produced by Unravel, were 'patched' by hand to produce the syntax-preserving slice used in the study.

5.4 Static Slicing: Results

There are 510 Lines of Code in the diff program, counted after the rewriting pre-transformations described in Section 5.3.1. The pseudo-variable, hp, is introduced in eight different places, to
model the effects of the malloc calls which allocate dynamic memory. Therefore, the subject program to be sliced for DMM-construction, contains 518 LoC for diff program. After slicing (with respect to hp at end of the main function), using Unravel, the LoC is reduced to 53 as shown in Table 5.8.

This slice does not include break statements and the body of the function add_exclude. These statements are required in order to form a valid syntax-preserving slice and so they were manually added. The LoC count for diff program, final, version of the syntax-preserving slice, is 64.

There have been few studies of the size of program slices for ‘typical’ programs and slicing criteria. One of the difficulties in studying slice sizes derives from the difficulty of selecting suitable criteria which are representative and in choosing an appropriate sample of the range of possible programs to which slicing might be applied. However, anecdotal evidence suggests that the size of syntax-preserving static slices tends to be disappointingly high. In one of the few formal studies for which detailed results are available, Venkatesh [78] studied dynamic slicing. Venkatesh found that the size of executable dynamic slices were about two thirds the size of the original programs from which they were constructed. Dynamic slices are always thinner than their corresponding static counterparts and so we can only suppose that syntax-preserving static slices will typically consist of more than two thirds of the program from which they are constructed.

Given previous results concerning the expected size of a ‘typical’ program slices (in general), the results obtained for the (specific) problem of dynamic memory modelling are surprisingly good. The syntax-preserving static slice is an order a magnitude smaller (518 LoC to 64 LoC) than the original program as shown in Table 5.8. Although this is only a single case (making generalisation difficult), we observe that dynamic memory is typically a relatively minor component of the overall computation of the program therefore, it is likely that slicing ought to be able to achieve a significant size reduction. This case study lends some evidence to support this claim.

Amorphous slicing further improves upon the performance of syntax-preserving slicing, reducing the size of the slice for all case studies. All the results are presented in Tables 5.8,
Using Conditioned Slicing

<table>
<thead>
<tr>
<th>Description</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rewritten version of <code>diff</code></td>
<td>510</td>
</tr>
<tr>
<td>Version with <code>hp</code> introduced</td>
<td>518</td>
</tr>
<tr>
<td>Unravel slice</td>
<td>53</td>
</tr>
<tr>
<td>Modified (final) syntax-preserving slice</td>
<td>64</td>
</tr>
<tr>
<td>Amorphous slice</td>
<td>52</td>
</tr>
</tbody>
</table>

Figure 5.8: LoC for static slicing of `diff` program

Figure 5.9: Original vs. Static Slices for `diff` program

5.14 and 5.18. The bar charts are shown in Figures 5.9, 5.15 and 5.19.

5.5 Using Conditioned Slicing

When we analysed the result of static slicing (both the syntax-preserving and amorphous forms), we have found out that the `diff` DMM slice had five separate and distinct behaviours, determined by the command line parameter switches provided by the user. It therefore seemed appropriate to consider the behaviour of the program in isolation, for each of these five behaviours. This is a simple form of conditioned slicing [19, 50].

A conditioned slice is constructed with respect to an augmented slicing criterion, which
additionally contains a predicate which determines an initial condition of interest [19]. Statements which cannot be executed when this condition is satisfied are removed, in a process called conditioning. Combining conditioning with static slicing leads to conditioned slicing.

For example, consider the program fragment in column (a) of Figure 5.10. The static slice of this program with respect to the variable $x$ is shown in column (b) of the figure, while the conditioned slice for $x$ where the initial condition is $x > 0$ is shown in column (c) of the figure.

Systems for fully conditioned slicing are currently at the ‘proof of concept level’ [25], and so it was not possible to use an fully-automated approach to the conditioned slicing of the `diff` program. However, the initial static slicing of the program revealed a top level structure (with respect to the heap computation) that was a simple case statement, with five cases. It was therefore possible to manually extract the slices for each of these five cases quite easily. This yielded five specialised programs, which captured the behaviour of the program for each of the five cases of interest\(^2\).

Once again we compared the size of the amorphous slices with the size of the syntax-preserving slices. The results are shown in Table 5.11 and Figures 5.12 and 5.13. The cases indicate the command-line option flags of interest. For some of the options (the F, I and p options) the slices are identical, and so these are depicted together. The `switch` statement also has a default case.

\(^2\)These are bona fide conditioned slices (rather than dynamic slices) because the two files, to which the `diff` program is applied, remain unspecified, though the command line switches are specified.
Using Conditioned Slicing

Conditioned slicing method is not applicable for Word count and DVD stock analysis program.

5.5.1 Conditioned Slicing: Results

As with the syntax-preserving and amorphous static slicing results, the use of conditioned slicing leads to a dramatic reduction in size, compared to the original program. Table 5.11 presents the results and Figure 5.12 shows these results as a histogram. The results for syntax-preserving slicing are sufficiently good so as to make visual comparison of syntax-preserving conditioned and amorphous conditioned slicing difficult. That is, it is hard to see the difference between amorphous and syntax-preserving slices because all are dwarfed by the size of the original program. To allow comparison of the slices alone (without the presence of the original program), Figure 5.13 shows the results for just amorphous and syntax-preserving conditioned slicing compared to the size of the static syntax-preserving and conditioned slices.

As can be seen from the results, syntax-preserving conditioned slicing further reduces the size of the syntax-preserving static slice, while amorphous conditioned slicing reduces this slice still further. Indeed, the amorphous conditioned slices constructed are commensurate with the size of a single function, ranging, as they do, from 12 to 35 lines of code. Each can easily be understood, by simple human examination. These results indicate that fully-automated amorphous conditioned slicing may be a great benefit to the analysis of dynamic memory modelling. The results of syntax-preserving static and conditioned slicing are sufficiently strong to suggest that even these larger forms of slice may be of significant assistance to dynamic memory analysis.
<table>
<thead>
<tr>
<th>Description</th>
<th>LoC for Amorphous Slice</th>
<th>LoC for syntax-preserving Conditioned Slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static version (all cases)</td>
<td>52</td>
<td>64</td>
</tr>
<tr>
<td>Case F, I, p</td>
<td>14</td>
<td>26</td>
</tr>
<tr>
<td>Case D</td>
<td>15</td>
<td>22</td>
</tr>
<tr>
<td>Case x</td>
<td>26</td>
<td>29</td>
</tr>
<tr>
<td>Case X</td>
<td>35</td>
<td>44</td>
</tr>
<tr>
<td>default</td>
<td>12</td>
<td>19</td>
</tr>
</tbody>
</table>

Figure 5.11: LoC calculation for Conditioned Slicing of diff program

Figure 5.12: Original vs. Conditioned Slicing for diff program
5.6 Word count program

Compared to the diff program, the Word count program is very small. To analyse this program we have used the prototype with some manual changes to the original program. Since the prototype does not cater for the memory model free, we have to add the pseudo variable statements for releasing memory by hand.

The complete source code has 35 LoC and introducing the pseudo variable, hp, increases this to 42. When we have done the syntax-preserving slice for the pseudo variable it reduced the LoC count to 10 LoC. Amorphous slicing reduced it further to 4 LoC as shown in Table 5.14. The graphical presentation is shown in Figure 5.15.
<table>
<thead>
<tr>
<th>Description</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word count program</td>
<td>35</td>
</tr>
<tr>
<td>Version with hp introduced</td>
<td>42</td>
</tr>
<tr>
<td>syntax-preserving slice</td>
<td>10</td>
</tr>
<tr>
<td>Amorphous slice</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 5.14: LoC for static slicing of Word count program

![Figure 5.14: LoC for static slicing of Word count program](image1)

The result $hp = 0$; shows clearly that there is no memory leak. Even though the Amorphous slice result contains only one assignment the LoC is 4. This is due to the way we measure the size of a program as explained in Section 5.3.1.

![Figure 5.15: Word count Original vs. Static Slices](image2)
Figure 5.16: Comparison of Static and Amorphous slicing for Word count program w.r.t. \texttt{hp}

As shown in Figure 5.16, the syntax-preserving slice clearly shows how many times the program has allocated and de-allocated the memory. According to the Amorphous slice result, we can deduce that the program has no memory leaks at all.

5.7 DVD stock analysis program

This is a small program, which allocates memory with the \texttt{for ... loop}. LoC and slice results are shown in Table 5.18. The graphical presentation is shown in Figure 5.19. With the original program, the user may not be able to identify the memory leaks at a glance. However, the Amorphous slice result shown in Figure 5.17 clearly indicates how much memory still has not been released.

Figure 5.17: Comparison of Static and Amorphous slicing for DVD stock analysis program w.r.t. \texttt{hp}
<table>
<thead>
<tr>
<th>Description</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DVD stock analysis program</td>
<td>38</td>
</tr>
<tr>
<td>Version with hp introduced</td>
<td>41</td>
</tr>
<tr>
<td>syntax-preserving slice</td>
<td>8</td>
</tr>
<tr>
<td>Amorphous slice</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 5.18: LoC for static slicing of Word count program

![Figure 5.19: DVD Stocking Analysis vs. Static Slices](image-url)
Chapter 6

Conclusions

This thesis introduces a new approach using the slicing technique to handle implicit-state-removal. It also demonstrates how Amorphous Program Slicing is used to create a Dynamic Memory Model (DMM) and presents 3 case studies to evaluate the use of slicing to construct the DMM of a program.

If a program needs to be sliced upon some value which is *implicit*, then the implicit-state-removal transformation technique can be applied. This technique is easy to integrate into existing slicing algorithms and tools, as only the defined and referenced variables need change. Furthermore, a suitable change in defined and referenced variables can be obtained by pre-transforming the program to be sliced to remove the implicit state.

We introduce a proof-of-concept DMM construction algorithm, showing how slicing simplification power can be dramatically improved using domain-specific transformation rules.

We illustrate that a DMM can be constructed using amorphous slicing on a program which has been pre-transformed into a version in which the top of the heap is explicit. The domain-specific nature of the approach means that powerful simplification transformations become applicable, making the size of DMMs surprisingly small.
Sometimes static slices tend to be rather large; often important computations touch a large section of the original program code. Amorphous slicing addresses this type of problem by focusing upon the semantics of the subcomponents to be ‘sliced away’ rather than the syntax in which they are expressed.

We also believe that typical programs will enjoy the smallest amount of syntax required to express some semantic feature with respect to the amount of dynamic memory allocated, making amorphous slicing highly applicable. This thesis has only considered a memory model in which memory is dynamically allocated in a C-like programming style.

We have examined the effect of syntax-preserving slicing and its amorphous counterpart in the analysis of dynamic memory allocation for the open source GNU diff program. Both amorphous static and conditioned forms of slices were constructed. The latter being constructed (partly) by hand (because conditioned slicing technology is still in its infancy, and therefore incapable of slicing a program, the size of diff).

The results are very encouraging. We have expected amorphous slicing to perform better than syntax-preserving slicing, since amorphous slices are (theoretically) always no larger than their syntax-preserving counterparts. However, even with syntax-preserving slicing, an order of magnitude size-reduction was achieved. This suggests a corresponding reduction in human effort required to analyse the dynamic memory characteristics of the original program.

For one of our case studies, our results were further improved by conditioned slicing, reducing the size of the program to be considered from 518 lines of code to between 12 and 35 lines of code, depending upon the condition chosen.
Chapter 7

Future Work

This thesis has primarily been concerned with memory allocation using malloc. It would be ideal to extend the work to consider look into functions like mark, release, free and realloc. These functions could be model in the following way:

7.1 mark

$E[mark(p)] = [p = hp]$  

The mark function marks a pointer’s current heap value.

<table>
<thead>
<tr>
<th>mark(p); hp = 0;</th>
<th>p = hp; a = mark(p);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>Implicit State Removed</td>
</tr>
</tbody>
</table>

Figure 7.1: Example of a mark

7.2 release

$E[release(p)] = [hp = p]$
The \texttt{release} function moves a pointer back to its marked position.

\begin{center}
\begin{tabular}{|c|c|}
\hline
\texttt{mark(p);} & \texttt{hp = 0;} \\
\texttt{malloc(32);} & \texttt{p = hp;} \\
\texttt{release(p);} & \texttt{a = mark(p);} \\
 & \texttt{hp = hp + 32;} \\
 & \texttt{malloc(32);} \\
 & \texttt{hp = p;} \\
 & \texttt{release(p);} \\
\hline
\end{tabular}
\end{center}

\begin{center}
<table>
<thead>
<tr>
<th>Original</th>
<th>Implicit State Removed</th>
</tr>
</thead>
</table>
\end{center}

Figure 7.2: Example of a \texttt{release}

### 7.3 \texttt{free}

\[ \mathcal{C}[\texttt{free(p):}] = [\texttt{hp = hp - StoreFor(p):}] \]

The \texttt{free} de-allocates the space pointed to by a pointer.

\begin{center}
\begin{tabular}{|c|c|}
\hline
\texttt{p = malloc(a);} & \texttt{hp = 0;} \\
\texttt{free(p);} & \texttt{hp = hp + a;} \\
 & \texttt{p = malloc(a);} \\
 & \texttt{hp = hp - a;} \\
 & \texttt{free(p);} \\
\hline
\end{tabular}
\end{center}

\begin{center}
<table>
<thead>
<tr>
<th>Original</th>
<th>Implicit State Removed</th>
</tr>
</thead>
</table>
\end{center}

Figure 7.3: Example of a \texttt{free}

### 7.4 \texttt{realloc}

\[ \mathcal{C}[\texttt{realloc(p,k):}] = [\texttt{hp = hp - k:}] \]
The `realloc` function de-allocate the space, which is specified by the second parameter, for a pointer. It is important to note that the functionality of this function is not same as ANSI ‘C’ compiler function.

<table>
<thead>
<tr>
<th>Original</th>
<th>Implicit State Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p = malloc(a);</code></td>
<td><code>hp = 0;</code></td>
</tr>
<tr>
<td><code>p = malloc(b);</code></td>
<td><code>hp = hp + a;</code></td>
</tr>
<tr>
<td><code>f = realloc(p, a);</code></td>
<td><code>p = malloc(a);</code></td>
</tr>
<tr>
<td></td>
<td><code>hp = hp + b;</code></td>
</tr>
<tr>
<td></td>
<td><code>p = malloc(b);</code></td>
</tr>
<tr>
<td></td>
<td><code>hp = hp - a;</code></td>
</tr>
<tr>
<td></td>
<td><code>f = realloc(p, a);</code></td>
</tr>
</tbody>
</table>

Figure 7.4: Example of a `realloc`

More work is required in order to implement and evaluate the application of these additional rules for memory modelling.

7.5 Assumptions

The thesis also contains a number of assumptions:

1. The function `malloc` can only be used on the right hand sides of assignment statement.

2. The actual parameter to `malloc` does not, itself, contain a call to `malloc`.

3. The data type of actual parameter is unsigned integer.

4. All the expressions are side effect free.

Future work could consider the effect of relaxing these assumptions.

In order to lift some of the assumptions in Section 7.5, we would be required to improve the slicing tool tremendously. This would also involve expanding the parser and the collapsing and transformation rules.
To implement the future work, the parser needs to be modified to be able to handle pointer variables. Also, new rules are required to defined functions like `mark`, `release`, `free` and `realloc` for collapsing and transformation rules.
Appendix A

Case Study - diff program and its slice results

A.1 Original diff program

The original program diff and related include files used in Chapter 5 - Case Study can be obtained from www.syssoft.co.uk/yoga/compsac/diff.tar

The original diff program can be access from www.syssoft.co.uk/yoga/compsac/exp01.c

A.2 Unravel version

The Unravel ready version can be access from www.syssoft.co.uk/yoga/compsac/exp02.c

This has 510 lines.

A.3 After the variable hp introduced

The pseudo variable, hp, introduced program can be access from www.syssoft.co.uk/yoga/compsac/exp03.c

This has 518 lines. The new lines shown in bold typeface.

```c
#define GDIFF_MAIN
ifndef DEFAULT_WIDTH
#define DEFAULT_WIDTH 130
endif
ifndef GUTTER_WIDTH_MINIMUM
#define GUTTER_WIDTH_MINIMUM 3
endif

int hp;

static int recursive;
int no_discards;
ifndef HAVE_SETMODE
```
static int binary_I_0;
#endif
static char *option_list (optionvec, count) char **optionvec;
/* Was 'vector', but that collides on Alliant. */ int count;
{
    int i;
    int length;
    char *result;
    length = 0;
    for (i = 0; i < count; i++)
        length += strlen (optionvec[i]) + 1;

hp = hp + (length + 1);

result = xmalloc (length + 1);
result[0] = 0;
for (i = 0; i < count; i++) {
    strcat (result, " ");
    strcat (result, optionvec[i]);
} return result;
}
static int ck_atoi (str, out) char const *str; int *out;
{
    char const *p;
    for (p = str; *p; p++)
        if (*p < '0' || *p > '9')
            return -1;
    *out = atoi (optarg);
    return 0;
}
static char const **exclude;
static int exclude_alloc, exclude_count;
int excluded_filename (f) char const *f;
{
    int i;
    for (i = 0; i < exclude_count; i++)
        if (fnmatch (exclude[i], f, 0) == 0)
            return 1;
    return 0;
}
static void add_exclude (pattern) char const *pattern;
{
    if (exclude_alloc <= exclude_count) {

        if (exclude_alloc == 0)
            exclude = (char const **) xmalloc ((exclude_alloc = 64) * sizeof (*exclude));
        else exclude = (char const **) realloc (exclude, (exclude_alloc *= 2) * sizeof (*exclude));
        exclude[exclude_count++] = pattern;
    }
static int add_exclude_file (name) char const *name;
{
    struct file_data f;
    char *p, *q, *lim;
A.3 After the variable hp introduced

```c
f.name = optarg;
if (strcmp(optarg, "-") == 0)
    f.desc = STDIN_FILENO;
else f.desc = open(optarg, O_RDONLY, 0);
if (f.desc < 0 || fstat(f.desc, &f.stat) != 0)
    return -1;
sip (&f, 1);
slurp (&f);
for (p = f.buffer, lim = p + f.buffered_chars; p < lim; p = q) {
    q = (char *) memchr(p, '
', lim - p);
    if (!q)
        q = lim;
    /*q++;*/
    add_exclude(p);
}
return close(f.desc);
}
static struct option const longopts[] =
{
    {"ignore-blank-lines", 0, 0, 'B'},
    {"context", 2, 0, 'C'},
    {"ifdef", 1, 0, 'D'},
    {"show-function-line", 1, 0, 'F'},
    {"speed-large-files", 0, 0, 'H'},
    {"ignore-matching-lines", 1, 0, 'I'},
    {"file-label", 1, 0, 'L'},
    {"context", 1, 0, 'S'},
    {"width", 1, 0, 'W'},
    {"text", 0, 0, 'a'},
    {"ignore-space-change", 0, 0, 'b'},
    {"minimal", 0, 0, 'd'},
    {"ed", 0, 0, 'e'},
    {"forward-ed", 0, 0, 'f'},
    {"ignore-case", 0, 0, 'i'},
    {"paginate", 0, 0, 'l'},
    {"print", 0, 0, 'p'},
    {"rcs", 0, 0, 'n'},
    {"show-c-function", 0, 0, 'p'},
    {"brief", 0, 0, 'q'},
    {"recursive", 0, 0, 'r'},
    {"report-identical-files", 0, 0, 's'},
    {"expand-tabs", 0, 0, 't'},
    {"version", 0, 0, 'v'},
    {"ignore-all-space", 0, 0, 'w'},
    {"exclude", 1, 0, 'x'},
    {"exclude-from", 1, 0, 'X'},
    {"side-by-side", 0, 0, 'y'},
    {"unified", 2, 0, 'U'},
    {"left-column", 0, 0, 'L'},
    {"suppress-common-lines", 0, 0, 130},
    {"sdiff-merge-assist", 0, 0, 131},
    {"old-line-format", 1, 0, 132},
};
```
int main (argc, argv) int argc; char *argv[];
{
    int val;
    int c;
    int prev;
    int width;
    int show_c_function;

    hp = 0;

    prev = -1;
    width = DEFAULT_WIDTH;
    show_c_function = 0;
    initialize_main (&argc, &argv);
    program_name = argv[0];
    output_style = OUTPUT_NORMAL;
    context = -1;
    while ((c = getopt_long (argc, argv, "123456789abBcC:dfD:efF:hi:Il:npqrs:stuU:wxy", longopts, 0)) != EOF) {
        switch (c) {
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
        case '0':
            if (context == -1)
                context = 0;
            else if (prev < '0' || prev > '9')
                fatal ("context length specified twice");
            context = context * 10 + c - '0';
        break;
        case 'a':
            always_text_flag = 1;
        break;
        case 'b':
            ignore_space_change_flag = 1;
            ignore_some_changes = 1;
        break;
case 'B':
    ignore_blank_lines_flag = 1;
    ignore_some_changes = 1;
    break;
case 'C': /* +context[=lines] */
case 'U': /* +unified[=lines] */
    if (optarg) {
        if (context >= 0)
            fatal ("context length specified twice");
        if (ck_atoi (optarg, &context))
            fatal ("invalid context length argument");
    }
case 'c':
    if (c == 'U')
        specify_style (OUTPUT_UNIFIED);
    else specify_style (OUTPUT_CONTEXT);
    break;
case 'd':
    no_discards = 1;
    break;
case 'D':
    specify_style (OUTPUTIFDEF);
    { int i, err;
      static char const C.ImageLayout_formats[] =
        "#ifndef %s
        #ifdef %s
        /* %s */\n        #endif
        /* %s */\n        #ifndef %s
        #ifdef %s
        /* %s */\n        #endif
        /* %s */\n        #ifndef %s
        #ifdef %s
        /* %s */\n        #endif
        /* %s */\n        #ifndef %s
        #ifdef %s
        /* %s */\n        #endif
        /* %s */\n        #ifndef %s
        #ifdef %s
        /* %s */\n        #endif
        /* %s */\n        #ifndef %s
        #ifdef %s
        /* %s */\n        #endif
        /* %s */\n    }
    err = 0;

    hp = hp + (sizeof (C.ImageLayout_formats) + 7 * strlen(optarg) - 14 - 8);
    *b = xmalloc (sizeof (CRoutingModule_formats) + 7 * strlen(optarg) - 14 /* strlen(optarg) - 8 */);
    sprintf (b, CRoutingModule_formats, optarg, optarg, 0, optarg, optarg, optarg, optarg);
    for (i = 0; i < 4; i++) {
        err |= specify_format (&group_format[i], b);
        b += strlen (b) + 1;
    }
    if (err)
        error ("conflicting #ifdef formats", 0, 0);
    break;
case 'e':
    specify_style (OUTPUT_ED);
    break;
case 'f':
    specify_style (OUTPUT_FORWARD_ED);
    break;
case 'F':
    add_regexp (&function_regexp_list, optarg);
    break;
case 'h':
    break;
case 'H':
    heuristic = 1;
    break;
case 'i':
    ignore_case_flag = 1;
    ignore_some_changes = 1;
ignore_some_line_changes = 1;
break;
case 'I':
    add_regexp (&ignore_regexp_list, optarg);
    ignore_some_changes = 1;
    break;
case 'L':
    paginate_flag = 1;
#if !defined(SIGCHLD) && defined(SIGCLD)
    #define SIGCHLD SIGCLD
#endif
#ifndef SIGCHLD
    signal (SIGCHLD, SIG_DFL);
#endif
break;
case 'L':
    if (!file_label[0])
        file_label[0] = optarg;
    else if (!file_label[1])
        file_label[1] = optarg;
    else
        fatal("too many file label options");
    break;
case 'N':
    specify_style (OUTPUT_RCS);
    break;
case 'N':
    entire_new_file_flag = 1;
    break;
case 'P':
    show_c_function = 1;
    add_regexp (&function_regexp_list, "[a-zA-Z]"); 
    break;
case 'P':
    unidirectional_new_file_flag = 1;
    break;
case 'Q':
    no_details_flag = 1;
    break;
case 'R':
    recursive = 1;
    break;
case 'S':
    print_file_same_flag = 1;
    break;
case 'S':
    dir_start_file = optarg;
    break;
case 'T':
    tab_expand_flag = 1;
    break;
case 'T':
    tab_align_flag = 1;
    break;
case 'U':
    specify_style (OUTPUT_UNIFORM);
    break;
A.3 After the variable `hp` introduced

```c
    case 'v':
        printf("diff - GNU diffutils version %s\n", version_string);
        exit (0);
    case 'w':
        ignore_all_space_flag = 1;
        ignore_some_changes = 1;
        ignore_some_line_changes = 1;
        break;
    case 'x':
        add_exclude(optarg);
        break;
    case 'X':
        if (add_exclude_file(optarg) != 0)
            pfatal_with_name(optarg);
        break;
    case 'y':
        specify_style (OUTPUT_SDIFF);
        break;
    case 'W':
        if (ck_atoi(optarg, &width) || width <= 0)
            fatal("column width must be a positive integer");
        break;
    case '29':
        sdiff_left_only = 1;
        break;
    case '30':
        sdiff_skip_common_lines = 1;
        break;
    case '31':
        specify_style (OUTPUT_SDIFF);
        sdiff_help_sdiff = 1;
        break;
    case '32':
    case '33':
    case '34':
        specify_style (OUTPUT_IFDEF);
        if (specify_format (&line_format[c - 132], optarg) != 0)
            error("conflicting line format". 0. 0);
        break;
    case '35':
        specify_style (OUTPUT_IFDEF);
        { int i, err;
          err = 0;
          for (i = 0; i < sizeof (line_format) / sizeof (*line_format); i++)
            err |= specify_format (&line_format[i], optarg);
          if (err)
              error("conflicting line format". 0. 0); }
        break;
    case '36':
    case '37':
    case '38':
    case '39':
        specify_style (OUTPUT_IFDEF);
        if (specify_format (&group_format[c - 136], optarg) != 0)
            error("conflicting group format". 0. 0);
        break;
    case '40':
```

if (d__atoi (optarg, &horizon_lines) || horizon_lines < 0)
    fatal ("horizon must be a nonnegative integer");
break;
case 141:
    usage ();
    check_stdout ();
    exit (0);
case 142:
#if HAVE_SETMODE
    binary_l_0 = 1;
    setmode (STDOUT_FILENO, O_BINARY);
#endif
break;
default:
    try_help (0); }
    prev = c; }
if (argc - optind != 2)
    if (argc - optind < 2)
        try_help ("missing operand");
    else try_help ("extra operand");
{ int t;
    int off;
    if (tab_expand_flag)
        t = 1;
    else t = TAB_WIDTH;
    off = (width + t + GUTTER_WIDTH_MINIMUM) / (2 * t) * t;
    sdiff_half_width = max (0, min (off - GUTTER_WIDTH_MINIMUM, width - off));
    if (sdiff_half_width)
        sdiff_column2_offset = off;
    else sdiff_column2_offset = width; }
if (show_c_function && output_style != OUTPUT_UNIFIED)
    specify_style (OUTPUT_CONTEXT);
if (output_style != OUTPUT_CONTEXT && output_style != OUTPUT_UNIFIED)
    context = 0;
else if (context == -1)
    context = 3;
if (output_style == OUTPUT_IFDEF) {
    int i;
    for (i = 0; i < sizeof (line_format) / sizeof (*line_format); i++)
        if (!line_format[i])
            line_format[i] = "%\n";
    if (!group_format[OLD])
        if (group_format[UNCHANGED])
            group_format[OLD] = group_format[UNCHANGED];
        else group_format[OLD] = "%<";
    if (!group_format[UNCHANGED])
        group_format[OLD] = group_format[UNCHANGED];
    else group_format[OLD] = "%<";
    if (!group_format[NEW])
        if (group_format[UNCHANGED])
            group_format[NEW] = group_format[UNCHANGED];
        else group_format[NEW] = "%";
    if (!group_format[UNCHANGED])
        group_format[UNCHANGED] = "%=";
    if (!group_format[UNCHANGED])
        group_format[UNCHANGED] = concat (group_format[OLD], group_format[NEW], ":");
    if (output_style == OUTPUT_IFDEF)
A.3 After the variable hp introduced

```c
if (!*group_format[UNCHANGED])
    no_diff_means_no_output = 1;
else if (strcmp (group_format[UNCHANGED], "\=\") == 0)
    no_diff_means_no_output = (*line_format[UNCHANGED]);
else no_diff_means_no_output = 0;
else if (*output_style == OUTPUT SDIFF)
    no_diff_means_no_output = sdiff_skip_common_lines;
else no_diff_means_no_output = 1;
switch_string = option_list (argv + 1, optind - 1);
val = compare_files (0, argv[optind], 0, argv[optind + 1], 0);
print_message_queue ();
check_stdout ();
exit (val);
}
```

```c
static void add_regexp (reglist, pattern) struct regexp_list **reglist; char const *pattern; {
    struct regexp_list *r;
    char const *m;

    hp = hp + (sizeof (*r));
    r = (struct regexp_list *) xmalloc (sizeof (*r));
    bzero (r, sizeof (*r));

    hp = hp + 256;
    r->buf.fastmap = xmalloc (256);
    m = re_compile_pattern (pattern, strlen (pattern), &r->buf);
    if (m != 0)
        error ("%s: %s", pattern, m);
    r->next = *reglist;
    *reglist = r;
}
```

```c
static void try_help (reason) char const *reason; {
    if (reason)
        error ("Try \"%s --help\" for more information.". program_name, 0);
    exit (2);
}
```

```c
static void check_stdout () {
    if (ferror (stdout) | fclose (stdout) != 0)
        fatal ("write error");
}
```

```c
static char const * const option_help[] = {
    "-i --ignore-case Consider upper- and lower-case to be the same.",
    "-w --ignore-all-space Ignore all white space.",
    "-b --ignore-space-change Ignore changes in the amount of white space.",
    "-B --ignore-blank-lines Ignore changes whose lines are all blank.",
    "-RE --ignore-matching-lines=RE Ignore changes whose lines all match RE.".
#if HAVE_SETMODE
    "--binary Read and write data in binary mode."
#endif
"-a --text Treat all files as text.\n"
```
"-c -C NUM --context=[=NUM] Output NUM (default 2) lines of copied context."
"-u -U NUM --unified=[=NUM] Output NUM (default 2) lines of unified context."
"-m Use NUM context lines."
"-L LABEL --label LABEL Use LABEL instead of file name.".
"-p --show-c-function Show which C function each change is in.".
"-f RE --show-function-line=RE Show the most recent line matching RE.".
"-q --brief Output only whether files differ.".
"-e --ed Output an ed script.".
"-n --rcs Output an RCS format diff.".
"-y --side-by-side Output in two columns.".
"-w NUM --width=NUM Output at most NUM (default 130) characters per line.".
"--left-column Output only the left column of common lines.".
"--suppress-common-lines Do not output common lines.".
"-DNAME --ifdef=NAME Output merged file to show `#ifdef NAME' diffs.".
"-GTYPE-group-format=GFMNT Similar, but format GTYPE input groups with GFMNT.".
"-l-line-format=LFMT Similar, but format all input lines with LFMT.".
"-LTYPE-line-format=LFMT Similar, but format LTYPE input lines with LFMT.".
" LTYPE is `old', `new', or `unchanged'. GTYPE is LTYPE or `changed'.".
" GFMNT may contain":
" %< lines from FILE1".
" %> lines from FILE2".
" %< lines common to FILE1 and FILE2".
" %<[WIDTH][.PREC][doxx]LETTER printf-style spec for LETTER".
" LETTERs are as follows for new group, lower case for old group:".
" F first line number".
" L last line number".
" N number of lines = L-F+1".
" E F-1".
" M L+1".
" LFMNT may contain":
" % contents of line".
" % contents of line, excluding any trailing newline".
" %<[WIDTH][.PREC]n printf-style spec for input line number".
" Either GFMNT or LFMNT may contain":
" % %".
" %c the single character C".
" %c'\000' the character with octal code 000\n".
"-l --paginate Pass the output through 'pr' to paginate it.".
"-t --expand-tabs Expand tabs to spaces in output.".
"-T --initial-tab Make tabs line up by prepending a tab.\n".
"-r --recursive Recursively compare any subdirectories found.".
"-N --new-file Treat absent files as empty.".
"-P --unidirectional-new-file Treat absent first files as empty.".
"-s --report-identical-files Report when two files are the same.".
"-x PAT --exclude-from=FILE Exclude files that match PAT.".
"-X FILE --exclude=FILE Exclude files that match any pattern in FILE.".
"-S FILE --starting-file=FILE Start with FILE when comparing directories.\n".
"-h --horizon-lines=NUM Keep NUM lines of the common prefix and suffix.".
"-d --minimal Try hard to find a smaller set of changes.".
"-H --speed-large-files Assume large files and many scattered small changes.\n".
"-w --version Output version info.".
"--help Output this help.".

};
static void usage ()
{
    const char * const *p;
A.3 After the variable hp introduced

printf ("Usage: %s [OPTION]... FILE1 FILE2\n\n", program_name);
for (p = option_help; *p; p++)
    printf ("%s\n", *p);
printf ("\nIf FILE1 or FILE2 is ‘-‘, read standard input.\n\n");
}
static int specify_format (var, value) char **var; char *value;
{
    int err;
    if (*var)
        err = strcmp (*var, value);
    else err = 0;
    *var = value;
    return err;
}
static void specify_style (style) enum output_style style;
{
    if (output_style != OUTPUT_NORMAL && output_style != style)
        error ("conflicting specifications of output style", 0, 0);
    output_style = style;
}
static char const *filetype (st) struct stat const *st;
{
    if (S_ISREG (st->st_mode)) {
        if (st->st_size == 0)
            return "regular empty file";
        return "regular file";
    }
    if (S_ISDIR (st->st_mode))
        return "directory";
#ifdef S_ISELK

    if (S_ISELK (st->st_mode))
        return "block special file";
#endif
#ifdef S_ISCHR

    if (S_ISCHR (st->st_mode))
        return "character special file";
#endif
#ifdef S_ISFIFO

    if (S_ISFIFO (st->st_mode))
        return "fifo";
#endif
#ifdef S_TYPEISPIMQ

    if (S_TYPEISPIMQ (st))
        return "message queue";
#endif
#ifdef S_TYPEISEM

    if (S_TYPEISEM (st))
        return "semaphore";
#endif
#ifdef S_TYPEISSIONM

    if (S_TYPEISSIONM (st))
        return "shared memory object";
#endif
#ifdef S_ISSOCK

    if (S_ISSOCK (st->st_mode))
        return "socket";
#endif
    return "weird file";
static int compare_files (dir0, name0, dir1, name1, depth)
    char const *dir0, *dir1; char const *name0, *name1; int depth;
{
    struct file_data inf[2];
    register int i;
    int val;
    int same_files;
    int failed;
    char *free0, *free1;
    failed = 0;
    *free0 = 0;
    *free1 = 0;
    if (! ((name0 != 0 && name1 != 0) || (unidirectional_new_file_flag && name1 != 0) || entire_new_file_flag)) {
        char const *name;
        char const *dir;
        if (name0 == 0)
            *name = name1;
        else *name = name0;
        if (name0 == 0)
            *dir = dir1;
        else *dir = dir0;
        message("Only in %s: %s\n", dir, name);
        return 1; }
    bzero (inf, sizeof (inf));
    if (name0 == 0)
        inf[0].desc = -1;
    else inf[0].desc = -2;
    if (name1 == 0)
        inf[1].desc = -1;
    else inf[1].desc = -2;
    if (name0 == 0)
        name0 = name1;
    if (name1 == 0)
        name1 = name0;
    if (dir0 == 0)
        inf[0].name = name0;
    else inf[0].name = (free0 = dir_file_pathname (dir0, name0));
    if (dir1 == 0)
        inf[1].name = name1;
    else inf[1].name = (free1 = dir_file_pathname (dir1, name1));
    for (i = 0; i < 1; i++) {
        if (inf[i].desc != -1) {
            int stat_result;
            if (i && filename_cmp (inf[i].name, inf[0].name) == 0) {
                inf[i].stat = inf[0].stat;
                stat_result = 0; }
            else if (strcmp (inf[i].name, "/") == 0) {
                inf[i].desc = STDIN_FILENO;
                stat_result = fstat (STDIN_FILENO, &inf[i].stat);
                if (stat_result == 0 && S_ISREG (inf[i].stat.st_mode)) {
                    if (pos == -1)
                        stat_result = -1;
                    else if (pos <= inf[i].stat.st_size)
                        inf[i].stat.st_size -= pos;
                    else inf[i].stat.st_size = 0; }
            else {
                if (stat_result == 0)
                    printf("%s\n", inf[i].name);
                else message("%s\n", inf[i].name); }
        } else {
            int stat_result = 0;
            if (inf[i].st_size > 0)
                printf("%s\n", inf[i].name);
            else message("%s\n", inf[i].name); }
    } else { /*error*/ }
A.3 After the variable hp introduced

time (&inf[i].stat.st_mtime); } } }
else stat_result = stat (inf[i].name, &inf[i].stat);
if (stat_result != 0) {
    perror_with_name (inf[i].name);
    failed = 1; }
else {
    inf[i].dir_p = S_ISDIR (inf[i].stat.st_mode) && inf[i].desc != 0;
    if (inf[1 - i].desc == -1) {
        inf[1 - i].dir_p = inf[i].dir_p;
        inf[1 - i].stat.st_mode = inf[i].stat.st_mode; } } }
if (*failed && depth == 0 && inf[0].dir_p != inf[1].dir_p) {
    int fnm_arg;
    int dir_arg;
    char const *fnm;
    char const *dir;
    char const *p;
    char const *filename;
    fnm_arg = inf[0].dir_p;
    dir_arg = 1 - fnm_arg;
    *fnm = inf[fnm_arg].name;
    *dir = inf[dir_arg].name;
    *p = filename_lastdirchar (fnm);
    if (p)
        inf[dir_arg].name = dir_file_pathname (dir, p + 1);
    else inf[dir_arg].name = dir_file_pathname (dir, fnm);
    *filename = inf[dir_arg].name;
    if (strcmp (fnm, "") == 0)
        fatal ("can't compare - to a directory");
    if (stat (filename, &inf[dir_arg].stat) != 0) {
        perror_with_name (filename);
        failed = 1; }
    else inf[dir_arg].dir_p = S_ISDIR (inf[dir_arg].stat.st_mode);
    if (failed) {
        val = 2; }
    else if ((same_files = inf[0].desc != -1 && inf[1].desc != -1
                 && 0 < same_files (&inf[0].stat, &inf[1].stat))
            && no_diff_means_no_output) {
        val = 0; }
    else if (inf[0].dir_p & inf[1].dir_p) {
        if (output_style == OUTPUTIFDEF)
            fatal ("-D option not supported with directories");
        if (depth > 0 && !recursive)
            message ("Common subdirectories: %s and %s\n", inf[0].name, inf[1].name);
        val = 0; }
    else {
        val = diff_dirs (inf.compare_files, depth); }
    else if ((inf[0].dir_p | inf[1].dir_p) || (depth > 0 && (! S_ISREG (inf[0].stat.st_mode)
                 || S_ISREG (inf[1].stat.st_mode)))) {
        if (inf[0].desc == -1 || inf[1].desc == -1) {
            if (inf[0].dir_p | inf[1].dir_p) && recursive
                && (entire_new_file_flag || (unidirectional_new_file_flag
                && inf[0].desc == -1)))
                val = diff_dirs (inf.compare_files, depth);
            else { char const *dir; if (inf[0].desc == -1)
                *dir = dir1;
            else *dir = dir0;
                message ("Only in %s: %s\n", dir, name0);
                val = 1; }
        } else { printf("Use -v for more debug output");
            val = 1; }
    }
else { message5 ("File %s is a %s while file %s is a %s\n", inf[0].name, filetype (&inf[0].stat), inf[1].name, filetype (&inf[1].stat));
    val = 1; }
else if ((no_details_flag & ~ignore_some_changes) &
    ((inf[0].stat.st_size != inf[1].stat.st_size)) &
    ((inf[0].desc == -1 || S_ISREG (inf[0].stat.st_mode)) &
        (inf[1].desc == -1 || S_ISREG (inf[1].stat.st_mode)))) {
    message ("Files %s and %s differ\n", inf[0].name, inf[1].name);
    val = 1; }
else { if (inf[0].desc == -2) &
    (inf[0].desc = open (inf[0].name, O_RDONLY, 0) < 0) {
        perror_with_name (inf[0].name);
        failed = 1; }
    if (inf[1].desc == -2) &
    (same_files) {
        if (inf[1].desc = open (inf[1].name, O_RDONLY, 0) < 0) {
            perror_with_name (inf[1].name);
            failed = 1; }
    &
    else if ((inf[1].desc = open (inf[1].name, O_RDONLY, 0) < 0) {
        perror_with_name (inf[1].name);
        failed = 1; }
    #if HAVE_SETMODE
    if (binary_I_0) &
    for (i = 0; i <= 1; i++) &
    if (0 <= inf[i].desc) &
        setmode (inf[i].desc, O_BINARY);
    #endif
    #endif
    if (failed) &
    val = 2; &
    else val = diff_2_files (inf, depth);
    if (inf[0].desc >= 0 && close (inf[0].desc) != 0) {
        perror_with_name (inf[0].name);
    val = 2; }
    if (inf[1].desc >= 0 && inf[0].desc != inf[1].desc && close (inf[1].desc) != 0) {
        perror_with_name (inf[1].name);
    val = 2; }
    if (val == 0 && !inf[0].dir_p) {
        if (print_file_same_flag)
            message ("Files %s and %s are identical\n", inf[0].name, inf[1].name);
        else fflush (stdout);
        if (free0)
            free (free0);
        if (free1)
            free (free1);
        return val;
    }#if HAVE_SETMODE
    if (binary_I_0)
        for (i = 0; i <= 1; i++)
            setmode (inf[i].desc, O_BINARY);
    #endif
    if (failed)
        val = 2;
    else val = diff_2_files (inf, depth);
    if (inf[0].desc >= 0 && close (inf[0].desc) != 0) {
        perror_with_name (inf[0].name);
    val = 2;
    if (inf[1].desc >= 0 && inf[0].desc != inf[1].desc && close (inf[1].desc) != 0) {
        perror_with_name (inf[1].name);
    val = 2;
    if (val == 0 && !inf[0].dir_p) {
        if (print_file_same_flag)
            message ("Files %s and %s are identical\n", inf[0].name, inf[1].name);
        else fflush (stdout);
        if (free0)
            free (free0);
        if (free1)
            free (free1);
        return val;
    }
    
A.4 All cases - Conditioned slice

This program can be obtained from www.sysssoft.co.uk/yoga/compsac/exp05.c

#define GDIFF_MAIN
#define DEFAULT_WIDTH
#define DEFAULT_WIDTH 130
#undef
#define GUTTER_WIDTH_MINIMUM
#define GUTTER_WIDTH_MINIMUM 3
#undef
int hp;
static int recursive;
int no_discards;
#endif
static int binary_I_0;
static int hp;
static char /*option_list (optionvec. count) char **optionvec;
#* Was `vector', but that collides on Alliant. */ int count;
{
    int i;
    int length;
    char *result;
    length = 0;
    for (i = 0; i < count; i++)
        length += strlen (optionvec[i]) + 1;
    hp = hp + (length + 1);
    result = xmalloc (length + 1);
    result[0] = 0;
    return result;
}
static char const /*/exclude;
static int exclude_alloc, exclude_count;
static void add_excluded_pattern (char const /*pattern;
{
    if (exclude_alloc <= exclude_count)
        if (exclude_alloc == 0)
            hp = hp + ((exclude_alloc = 64) * sizeof (*exclude));
        else hp = hp + ((exclude_alloc == 2) * sizeof (*exclude)) - sizeof(*exclude);
        exclude[exclude_count++] = pattern;
}
static int add_excluded_file (name) char const /*name;
{
    struct file_data f;
    char p, q, lim;
    if (f.desc < 0 || fstat (f.desc, &f.stat) != 0)
        return -1;
    for (p = f.buffer, lim = p + f.buffered_chars; p < lim; p = q) {
        q = (char *) memchr (p, '
', lim - p);
        if (!q)
            q = lim;
        *q++ = 0;
        add_excluded (p); }
    return close (f.desc);
}
int main (argc, argv) int argc; char *argv[];
{
    hp = 0;
    initialize_main (&argc, &argv);
    while ((c = getopt_long (argc, argv, "D:abcdef:hlmpnrstTuvwx\", longopts, 0)) != EOF) {
        switch (c) {
        case 'D':
            { static char const C_ifdef_group_formats[] =
               "#ifndef \%s\n#ifndef \%s#endif /* not \%s */\%s#endif /* \%s */\%s#endif
               /* \%s */\%s#endif
               /* \%s */\%s#endif
               /* \%s */\%s#endif
               /* \%s */\%s#endif
               /* \%s */\%s#endif
               /* \%s */
               hp = hp + (sizeof (C_ifdef_group_formats) + 7 * strlen(optarg) - 14 - 8 );
            break;
            }
case 'F':
    add_regex (&function_regex_list, optarg);
    break;
case 'I':
    add_regex (&ignore_regex_list, optarg);
    break;
case 'p':
    add_regex (&function_regex_list, "[^a-zA-Z]" );
    break;
case 'x':
    add_exclude(optarg);
    break;
case 'X':
    if (add_exclude_file(optarg) != 0)
        ;
    break;
default:}
    switch_string = option_list (argv + 1, optind - 1);
}
static void add_regex (reglist, pattern) struct regexp_list *reglist; char const *pattern;
{
    struct regexp_list *r;
    hp = hp + (sizeof (*r));
    hp = hp + 256;
}

A.5 All cases - Amorphous slice

This program can be obtained from www.sysoff.co.uk/yoga/compsac/exp06.c

#define GDIFF_MAIN
#ifndef DEFAULT_WIDTH
#define DEFAULT_WIDTH 130
#endif
#ifndef GUTTER_WIDTH_MINIMUM
#define GUTTER_WIDTH_MINIMUM 3
#endif
    int hp;
static int recursive;
int no_discards;
#ifndef HAVE_SETMODE
    static int binary_I_0;
#endif
static char const *exclude;
static int exclude_alloc, exclude_count;
static void add_exclude (pattern) char const *pattern;
{
    if (exclude_alloc != exclude_count) {
        if (exclude_alloc == 0)
            hp = hp + (exclude_alloc = 64) * sizeof (*exclude));
        else hp = hp + (exclude_alloc == 2) * sizeof (*exclude)) - sizeof (*exclude);
    exclude[exclude_count++] = pattern;
}
static int add_exclude_file (name) char const *name;
{
    struct file_data f;
    char *p, *q, *lim;
A.6 Case F, I, p - Conditioned slice

This program can be obtained from www.syssoft.co.uk/yoga/compsac/exp08a.c
#endif
int hp;
static int recursive;
int no_discards;
#if HAVE_SETCMODE
static int binary_I_0;
#endif
static char *option_list (optionvec, count) char **optionvec;
/* Was 'vector', but that collides on Alliant. */ int count;
{
    int i;
    int length;
    char *result;
    length = 0;
    for (i = 0; i < count; i++)
        length += strlen (optionvec[i]) + 1;
    hp = hp + (length + 1);
    result = xmalloc (length + 1);
    result[0] = 0;
    return result;
}
int main (argc, argv) int argc; char *argv[];
{
    hp = 0;
    initialize_main (&argc, &argv);
    while ((c = getopts_long (argc, argv, "0123456789abCcdEdEdFgHiIlLmmpqrsStTuUvwWxXy",
                                longopts, 0) != EOF) {
        switch (c) {
            case 'F':
                add_regexp (&function_regexp_list, optarg);
                break;
            case 'I':
                add_regexp (&ignore_regexp_list, optarg);
                break;
            case 'p':
                add_regexp (&function_regexp_list, ".[^-a-zA-Z]" );
                break; }
            switch_string = option_list (argv + 1, optind - 1);
        }
    }
    static void add_regexp (reglist, pattern) struct regexp_list **reglist; char const *pattern;
    {
        struct regexp_list *r;
        hp = hp + (sizeof (*r));
        hp = hp + 256;
    }

A.7 Case F, I, p - Amorphous slice

This program can be obtained from www.syssoft.co.uk/yoga/compsac/exp07a.c

#define GDIFF_MAIN
#define DEFAULT_WIDTH
#define DEFAULT_WIDTH 130
#endif
#define GUTTER_WIDTH_MINIMUM
#define GUTTER_WIDTH_MINIMUM 3
#define GUTTER_WIDTH_MINIMUM 3

A.7 Case F, I, p - Amorphous slice

This program can be obtained from www.syssoft.co.uk/yoga/compsac/exp07a.c

#define GDIFF_MAIN
#define DEFAULT_WIDTH
#define DEFAULT_WIDTH 130
#endif
#define GUTTER_WIDTH_MINIMUM
#define GUTTER_WIDTH_MINIMUM 3
#define GUTTER_WIDTH_MINIMUM 3
A.8 Case D - Conditioned slice

This program can be obtained from www.syssoft.co.uk/yoga/compsac/exp08b.c
A.9 Case D - Amorphous slice

This program can be obtained from www.syssoft.co.uk/yoga/compsac/exp07b.c

#define GDIFF_MAIN
#ifndef DEFAULT_WIDTH
#define DEFAULT_WIDTH 130
#endif
#ifndef GUTTER_WIDTH_MINIMUM
#define GUTTER_WIDTH_MINIMUM 3
#endif

int hp;
static int recursive;
int no_discards;
#if HAVE_SETMODE
static int binary_I_0;
#endif

int main (argc, argv) int argc; char *argv[];
{
    hp = 0;
    initialize_main (&argc, &argv);
    while ((c = getopt_long (argc, argv, "0123456789abBcCdEfHhiIl:mnOpQrStTuUvwxXY", longopts, 0)) != EOF) {
        switch (c) {
            case 'D':
                static char const C ifdef_group_formats[] =
                    
                hp = hp + (sizeof (C ifdef_group_formats) + 7 * strlen (optarg) - 14 - 8);
            break;
            }
            { int i; int length;
            length = 0;
            for (i = 0; i < (optind - 1); i++)
                length += strlen (argv[i + 1]) + 1;
            hp = hp + (length + 1);
            }
    }
}

A.10 Case x - Conditioned slice

This program can be obtained from www.syssoft.co.uk/yoga/compsac/exp08c.c

#define GDIFF_MAIN
#ifndef DEFAULT_WIDTH
#define DEFAULT_WIDTH 130
#endif
#ifndef GUTTER_WIDTH_MINIMUM
#define GUTTER_WIDTH_MINIMUM 3
#endif

int hp;
static int recursive;
int no_discards;
#if HAVE_SETMODE
static int binary_I_0;
#endif

static char *option_list (optionvec, count) char **optionvec;
/* Was 'vector', but that collides on Alliant. */ int count;
{
```c
int i;
int length;
char *result;
length = 0;
for (i = 0; i < count; i++)
    length += strlen(optionvec[i]) + 1;
hp = hp + (length + 1);
result = xmalloc(length + 1);
result[0] = 0;
return result;
}
static char const **exclude;
static int exclude_alloc, exclude_count;
static void add_exclude (pattern) char const *pattern;
{
    if (exclude_alloc <= exclude_count) {
        if (exclude_alloc == 0)
            hp = hp + ((exclude_alloc = 64) * sizeof(*exclude));
        else hp = hp + ((exclude_alloc == 2) * sizeof(*exclude)) - sizeof(*exclude);
    }
    exclude[exclude_count++] = pattern;
}
int main (argc, argv) int argc; char *argv[];
{
    hp = 0;
    initialize_main (&argc, &argv);
    while ((c = getopt_long(argc, argv, "0123456789abcCd:efhi:kl:nopqrstuvwxxy", longopts, 0)) != EOF) {
        switch (c) {
        case 'x':
            add_exclude(optarg);
            break;
        }
        switch_string = option_list (argv + 1, optind - 1);
    }
```

A.11 Case x - Amorphous slice

This program can be obtained from www.syssoft.co.uk/yoga/compsac/exp07c.c
A.12 Case X - Conditioned slice

This program can be obtained from www.syssoft.co.uk/yoga/compsac/exp08d.c
A.13 Case X - Amorphous slice

This program can be obtained from www.syssoft.co.uk/yoga/compac/exp07d.c

```c
if (exclude_alloc == 0)
    hp = hp + ((exclude_alloc = 64) * sizeof (*exclude));
else hp = hp + ((exclude_alloc == 2) * sizeof (*exclude)) - sizeof(*exclude);
exclude[exclude_count++] = pattern;
}
static int add_exclude_file (name) char const *name;
{
    struct file_data f;
    char *p, *q, *lim;
    if (f.desc < 0 || fstat (f.desc, &f.stat) != 0)
        return -1;
    for (p = f.buffer, lim = p + f.buffered_chars; p < lim; p = q) {
        q = (char *) memchr (p, '\n', lim - p);
        if (!q)
            q = lim;
        *q++ = 0;
        add_exclude (p); }
    return close (f.desc);
}
static int recursive;
int no_discards;
#if HAVE_SETMODE
static int binary_I_O;
#endif
static char const **exclude;
static int exclude_alloc, exclude_count;
static void add_exclude (pattern) char const *pattern;
{
    if (exclude_alloc == exclude_count) {
        if (exclude_alloc == 0)
            hp = hp + ((exclude_alloc = 64) * sizeof (*exclude));
```


```c
else hp = hp + ((exclude_alloc == 2) * sizeof (*exclude)) - sizeof(*exclude);
exclude[exclude_count++] = pattern;
}
static int add_exclude_file (name) char const *name;
{
    struct file_data f;
    char *p, *q, *lim;
    if (f.desc < 0 || fstat (f.desc, &f.stat) != 0)
        return -1;
    for (p = f.buffer, lim = p + f.buffered_chars; p < lim; p = q) {
        q = (char *) memchr (p, '\n', lim - p);
        if (!q)
            q = lim;
        *q++ = 0;
        add_exclude (p);
    }
    return close (f.desc);
}
int main (argc, argv) int argc; char *argv[];
{
    hp = 0;
    initialize_main (&argc, &argv);
    while ((c = getopt_long (argc, argv, "0123456789abCdD:eFfHhiIl:rnsp:tuwxy",
                             longopts, 0)) != EOF) {
        switch (c) {
            case 'X':
                if (add_exclude_file (optarg) != 0)
                    break;
        }
        { int i; int length;
            length = 0;
            for (i = 0; i < (optind - 1); i++)
                length += strlen (argv[i + 1]) + 1;
            hp = hp + (length + 1);
        }
}

A.14 Case default - Conditioned slice

This program can be obtained from www.syssoft.co.uk/yoga/compsac/exp08e.c

#define GDIFF_MAIN
#ifndef DEFAULT_WIDTH
#define DEFAULT_WIDTH 130
#endif
#ifndef GUTTER_WIDTH_MINIMUM
#define GUTTER_WIDTH_MINIMUM 3
#endif
int hp;
static int recursive;
int no_discards;
#if HAVE_SETMODE
static int binary_I_0;
#endif
static char *option_list (optionvec, count) char **optionvec;
/* Was 'vector', but that collides on Alliant. */ int count;
{
    int i;
```
int length;
char *result;
length = 0;
for (i = 0; i < count; i++)
  length += strlen (optionvec[i]) + 1;
hp = hp + (length + 1);
result = xmalloc (length + 1);
result[0] = 0;
return result;
}
int main (argc, argv) int argc; char *argv[];
{
  hp = 0;
  initialize_main (&argc, &argv);
  while ((c = getopt_long (argc, argv, "", longopts, 0)) != EOF) {
    switch (c) {
    default:;
    }
    switch_string = option_list (argv + 1, optind - 1);
  }
}

A.15 Case default - Amorphous slice

This program can be obtained from www.syssof.co.uk/yoga/compsac/exp07e.c

#define GDIF_MAIN
#ifndef DEFAULT_WIDTH
#define DEFAULT_WIDTH 130
#endif
#ifndef GUTTER_WIDTH_MINIMUM
#define GUTTER_WIDTH_MINIMUM 3
#endif
int hp;
static int recursive;
int no_discards;
#if HAVE_SETMODE
static int binary_I_0;
#endif
int main (argc, argv) int argc; char *argv[];
{
  hp = 0;
  initialize_main (&argc, &argv);
  while ((c = getopt_long (argc, argv, "", longopts, 0)) != EOF) {
    switch (c) {
    default:;
    }
    { int i; int length;
      length = 0;
      for (i = 0; i < (optind - 1); i++)
        length += strlen (argv[i + 1]) + 1;
      hp = hp + (length + 1);
    }
Case Study - diff program and its slice results
Appendix B

Case Study - Word count program
and its slice results

B.1 Word count program

This has 35 lines.

#include <stdio.h>

main()
{
    char *strFilename, *strLineBuffer, *strTemp;
    char *LargestWord;
    FILE *infile;
    int intLineNo, intLineLength;
    int k, intNoOfWords;

    strFilename = malloc(100);
    strLineBuffer = malloc(500);
    strTemp = malloc(500);
    printf("Please enter a file name > ");
    scanf("%s", strFilename);
    printf("The Input Filename: %s\n\n", strFilename);
    printf("Line Length No Of String\n\n", strFilename);
    printf("No Words\n\n", strFilename);
    printf("--------------------------\n\n", strFilename);

    if (fopen(strFilename, "rt") == NULL) {
        printf("*** ERROR: Failed to open source file.\n\n");
        return;
    }
    infile = fopen(strFilename, "rt");

    intLineNo = 0;
    intNoOfWords = 0;
    intLineNo = intLineNo + 1;
    intLineLength = 0;

    for (k=0; strLineBuffer != '0'; k = k + 1) {
        strTemp = strLineBuffer;
        strTemp := strTemp + 1;
strLineBuffer := strLineBuffer + 1;
}

if (strLineBuffer == ' ')
    intNoOfWords = intNoOfWords + 1;
intLineLength = intLineLength + 1;
strLineBuffer = strLineBuffer + 1;
if (strLineBuffer == '
')
    intNoOfWords = intNoOfWords + 1;
printf("%5d %6d %5d %s", intLineNo, intLineLength, intNoOfWords, strTemp);
fclose(infile);
free(strFilename);
free(strLineBuffer);
free(strTemp);
}

B.2 After the variable hp introduced

This has 42 lines.

main()
{
    hp = 0;
    hp = hp + 100;
    strFilename = malloc(100);
    hp = hp + 500;
    strLineBuffer = malloc(500);
    hp = hp + 500;
    strTemp = malloc(500);
    printf("Please enter a file name > ");
    scanf("%s", strFilename);
    printf("The Input Filename: %s\n", strFilename);
    printf("Line Length No Of String \n", strFilename);
    printf("No Words \n", strFilename);
    printf("\n ---------------\n", strFilename);
    if (fopen(strFilename, "rt") == NULL) {
        printf("*** ERROR: Failed to open source file.\n");
        return;
    }
    infile = fopen(strFilename, "rt");

    intLineNo = 0;
    intNoOfWords = 0;
    intLineNo = intLineNo + 1;
    intLineLength = 0;
    for (k=0; strLineBuffer != '>'; k = k + 1) {
        strTemp = strLineBuffer;
B.3 Syntax-preserving slice

This has 10 lines.

main()
{
    hp = 0;
    hp = hp + 100;
    hp = hp + 500;
    hp = hp + 500;
    hp = hp - 100;
    hp = hp - 500;
    hp = hp - 500;
}

B.4 Amorphous slice

This has 4 lines.

main()
{
    hp = 0;
}
Appendix C

Case Study - DVD record count program and its slice results

C.1 DVD record count program

This has 38 lines.

```
#include <stdio.h>

typedef struct dvdstruc *DVDRecs;
struct dvdstruc {
    char Title[30];
    int NoOfCopies;
    int Status;
};

typedef struct liststruct* list;
struct liststruct {
    void* datum;
    list next;
};

main()
{

    int NumberOfDVDs, TotalDVDs, InStockDVDs, j, k, Copies, Status;
    char Title[30];
    DVDRecs DVDs;
    list DVDList; r;

    printf("Number of DVDs?\n");
    scanf("%d", &NumberOfDVDs);
    TotalDVDs = 0;
    DVDList = NULL;

    for (k=0; k<NumberOfDVDs; k=k+1) {
        DVDs = (DVDRecs) malloc(34);
        printf("\n%*d DVD Title Name: %s", k);
        scanf("%*s", Title);
        strcpy(DVDs->Title, Title);
        printf("\n%*d No of Copies: %s", k);
        ...
Case Study - DVD record count program and its slice results

```c
scanf("%d", &Copies);
DVDs->NoOfCopies = Copies;

TotalDVDs = TotalDVDs + Copies;
DVDs->Status = 0;

for (j=0; j<Copies; j+=1) {
    printf("Title: %s, Copy: %d Status: ", Title, j+1);
    scanf("%d", &Status);

    if (Status==1)
        DVDs->Status = Status;
}

r = (list) malloc(4);
r->datum = DVDs;
r->next = DVDList;
DVDList = r;
}

InStockDVDs = 0;
for (k=0; k<NumberOfDVDs; k+=1) {
    DVDs = DVDList->datum;

    printf("%s", DVDs->Title);
    printf("%d", DVDs->NoOfCopies);

    for (j=0; j<Copies; j+=1) {
        printf("Title: %s, Copy: %d Status: %d ", DVDs->Title, DVDs->NoOfCopies, DVDs->Status);
    }

    if (DVDs->Status == 1)
        InStockDVDs = InStockDVDs + 1;
    DVDList = DVDList->next;
}

printf("\n\nTotal number of DVDs including copies: %d", TotalDVDs);
printf("\nNumber of DVD titles are in stock: %d", InStockDVDs);
}
```

C.2 After the variable hp introduced

This has 41 lines.

```c
main()
{
    hp = 0;

    printf("Number of DVDs?\n");
    scanf("%d", &NumberOfDVDs);
    TotalDVDs = 0;
    DVDList = NULL;
```
C.3 Syntax-preserving slice

```c
for (k=0; k<NumberOfDVDs; k=k+1) {
    hp = hp + 34;
    DVDs = malloc(34);
    printf("%d DVD Title Name: ",k+1);
    scanf("%s",&Title);
    strcpy(DVDs->Title, Title);
    printf("%d No of Copies: ",k+1);
    scanf("%d",&Copies);
    DVDs->NoOfCopies = Copies;
    TotalDVDs = TotalDVDs + Copies;
    DVDs->Status = 0;

    for (j=0; j<Copies; j=j+1) {
        printf("Title:/%s Copy:%d Status/: ",Title, j+1);
        scanf("%d",&Status);
        if (Status==1)
            DVDs->Status = Status;
    }
    hp = hp + 4;
    r = malloc(4);
    r->datum = DVDs;
    r->next = DVDList;
    DVDList = r;
}

InStockDVDs = 0;
for (k=0; k<NumberOfDVDs; k=k+1) {
    DVDs = DVDList->datum;
    printf("%s ",DVDs->Title);
    printf("%d",DVDs->NoOfCopies);
    for (j=0; j<Copies; j=j+1) {
        printf("Title:/%s Copy:%d Status:%d ",DVDs->Title, DVDs->NoOfCopies, DVDs->Status);
    }
    if (DVDs->Status == 1)
        InStockDVDs = InStockDVDs + 1;
    DVDList = DVDList->next;
}
printf("Total number of DVDs including copies: %d 

Number of DVD titles in stock: %d 

Toatal number of DVDs including copies: %d 

Number of DVD titles are in stock: %d
```

C.3 Syntax-preserving slice

This has 8 lines.
C.4 Amorphous slice

This has 5 lines.

```c
hp = 0;
scanf("%d", &NumberOfDVDs);
for (k=0; k<NumberOfDVDs; k=k+1) {
    hp = hp + 34;
    hp = hp + 4;
}
main()
{
    scanf("%d", &NumberOfDVDs);
    hp = (NumberOfDVDs) * 38;
}```
Bibliography


