A Non-Standard Semantics for Program Slicing and Dependence Analysis

By

Lahcen Ouarbya

This thesis is submitted in fulfilment of the requirements for the degree of Doctor Of Philosophy in the

Department of Computing

Goldsmiths College

University of London

New Cross, London SE14 6NW, UK.

© 2005 Lahcen Ouarbya
Abstract

The aim of this thesis is to investigate slicing without intermediate graph structures where
the slicing algorithm and the semantics of the program language are both expressed de-
notationally. This should allow the possibility of using the full power and elegance of
denotational semantics in definitions and correctness proofs.

1. We show that neededness (Variable Dependence), an essential idea in slicing, cannot
   be expressed naturally using standard denotational semantics.

2. As a result of this, a non-standard lazy denotational semantics of an imperative
   language is introduced. This semantics, is different from other lazy semantics used
   in static program analysis in that it is substitutive thereby rendering it more useful
   for proving correctness of transformation systems where a program’s transformations
   are expressed in terms of the transformations of its sub-components.

3. The applicability of our semantics is demonstrated by proving the correctness of
   Hausler’s Denotational program slicing algorithm.
Dedication

I dedicate this dissertation to my wife, Sarah, my son Samir and to the memory of my parents. I will be forever grateful for the love, support and guidance you have given me throughout my life. God bless you all.
Acknowledgements

First of all, and most importantly, I would like to thank my supervisors, Dr. Sebastian Danicic and Dr. John Howroyd, for facilitating my journey and for the advice, support and inspiration that they have given me on the way. Without you I would not have reached the position that I have today. I would also like to give special recognition and thanks to Pr. Mark Harman of Kings College and Dr. Chris Fox of Essex university, my external supervisors, for providing me with valuable advice, guidance and encouragement whilst writing this dissertation. Your time and patience are greatly appreciated.

Thanks also to Rob Hierons, Lin Hu, Dave Binkley, Martin Ward, Roberto Giacobazzi and Isabella Mastroeni. Your comments have been very useful to me.

I would like to thank my wife, Sarah, for all her love, unconditional support and understanding during what has sometimes been a difficult journey. My thanks are also extended to my in-laws, Ian and Glenda Pacey, thank you for welcoming me into your family and for the belief that you have shown in me. I am also indebted to my friend, Mohammed Daoudi, for his friendship and assistance over the four years that I have known him.

I am grateful to the Department of Computing of Goldsmiths College, University of London for giving me the opportunity and a bursary to embark on this project. Goldsmiths has provided an excellent environment for research and I acknowledge with gratitude the help and support provided by its staff.

Last, but far from least, I would like to give my sincere thanks to all of my family and my friends for their emotional support, patience and understanding. You are all fantastic.
Publications
Publications
declaration

I hereby declare that I composed this thesis entirely by myself and that it describes my own research.

Lahcen Ouarbya
Department of Mathematical and Computing Sciences
Goldsmiths College, London University
May 11, 2005
Contents

Abstract

List of Contents

List of Figures

1 Introduction

2 Background

2.1 Control Flow .......................................................... 22
  2.1.1 Control Flow Graphs ......................................... 22
  2.1.2 Post-dominance in the Control-Flow Graph ............... 23
  2.1.3 Control Dependences ......................................... 25

2.2 Data Dependence ..................................................... 27
  2.2.1 Computation of Used(Referenced) and Defined Variables .... 27
  2.2.2 Computation of Reaching Definitions ....................... 29

2.3 Program dependence graph ...................................... 34

2.4 Program Slicing .................................................... 36
  2.4.1 Different Types of Program Slicing ....................... 36

2.5 Current slicing algorithms ...................................... 44
  2.5.1 Weiser’s slicing algorithm .................................. 44

Abstract

List of Contents

List of Figures

1 Introduction

2 Background

2.1 Control Flow .......................................................... 22
  2.1.1 Control Flow Graphs ......................................... 22
  2.1.2 Post-dominance in the Control-Flow Graph ............... 23
  2.1.3 Control Dependences ......................................... 25

2.2 Data Dependence ..................................................... 27
  2.2.1 Computation of Used(Referenced) and Defined Variables .... 27
  2.2.2 Computation of Reaching Definitions ....................... 29

2.3 Program dependence graph ...................................... 34

2.4 Program Slicing .................................................... 36
  2.4.1 Different Types of Program Slicing ....................... 36

2.5 Current slicing algorithms ...................................... 44
  2.5.1 Weiser’s slicing algorithm .................................. 44
2.5.2 Slicing as a Graph-Reachability Problem .......................... 48
2.6 Applications ........................................................................ 57
  2.6.1 Debugging ................................................................. 57
  2.6.2 Program Differencing .................................................... 61
  2.6.3 Program Integration ...................................................... 62
  2.6.4 Software Maintenance ................................................... 63
  2.6.5 Testing ................................................................. 64
2.7 Program Semantics and Program Slicing ............................... 65
  2.7.1 Standard Denotational Semantics ................................. 65
  2.7.2 Weakest Precondition and Strongest Postcondition Semantics .... 67
  2.7.3 Lazy semantics of program dependence graphs—Cartwright and Felleisen .................................................. 74
  2.7.4 Transfinite semantics—Giacobazzi and Mastroeni .............. 80
  2.7.5 Operational Semantics ................................................ 83

3 Neededness—Variable Dependence ........................................... 87
  3.1 Hausler’s Slicing Algorithm .............................................. 87
    3.1.1 Hausler Needed—$N_h$ ........................................... 87
    3.1.2 The Algorithm for $N_h$ .......................................... 90
    3.1.3 Hausler’s Slicing Algorithm—$S_h$ ............................. 91
  3.2 Neededness ................................................................. 93
    3.2.1 Partially Discriminating ........................................... 94
    3.2.2 Hausler—consistent ................................................. 94
    3.2.3 Sub-sequentiality of Neededness ............................... 95
  3.3 Neededness and Standard Semantics ................................... 96
3.3.1 Unfolding of while Loops ........................................ 97
3.3.2 Semantic Discrimination .......................................... 99

4 Lazy Semantics for Program Slicing ................................. 101
  4.1 Lazy Semantics ..................................................... 101
       4.1.1 Properties of the Lazy Semantics ......................... 109
       4.1.2 Lazy Semantic Discrimination Relation .................. 112
       4.1.3 Lazy Semantics is Substitutive ......................... 113
  4.2 Defining Slices using Lazy Semantics ......................... 114
       4.2.1 Properties of $\leq$ ........................................ 117
  4.3 Lazy Neededness .................................................. 118
       4.3.1 Lazy Neededness is Sub-sequential ...................... 124
       4.3.2 Lazy Neededness is Consistent with Hausler-Neededness 125

5 Correctness Proof of Hausler’s Slicing Algorithm .................. 129
  5.1 Lazy Semantics is Preserved by Hausler’s Algorithm .......... 129
  5.2 Termination Neededness .......................................... 134
  5.3 Hausler’s Slicing Algorithm Preserves Termination ............. 140

6 A Denotational Interprocedural Program Slicer ..................... 145
  6.1 Slicing in the Presence of side-effects ....................... 146
  6.2 Interprocedural Slicing ......................................... 147
       6.2.1 Slicing Expressions .................................... 149
       6.2.2 Interprocedural Slicing of Statements ................. 153
  6.3 Related Work ................................................... 156
       6.3.1 Advantages of Our Approach ............................ 157
7 Conclusion and Future Work 158

7.1 Conclusion ................................................................. 158
7.2 Future work ................................................................. 160
  7.2.1 Extending the Lazy semantics to Programs with Procedures . . . . 160
  7.2.2 Extending Hausler's slicing algorithm to Programs with Procedures 167

Bibliography 169
# List of Figures

1.1 A simple program $P$ .......................................................... 17

1.2 Transfinite meaning of $y$ is $\omega$ ................................. 18

1.3 $P_1$ and $P_2$ do not have the same transfinite semantics w.r.t. $x$. .............................. 19

2.1 Sum program $P$ ................................................................. 23

2.2 CFG of the program in Figure 2.1 where Def and Ref are the set of variables defined and referenced at a program point. .................................................. 24

2.3 Control dependencies of the program in Figure 2.1 .............................. 26

2.4 Data-flow dependencies of the program in Figure 2.1 .............................. 28

2.5 Used and defined variables .................................................... 29

2.6 Example illustrating $gen$ and $kill$ ....................................... 29

2.7 Data-flow equations for an assignment ..................................... 30

2.8 Data-flow equations for sequence of statements—($S1; S2$) .................. 32

2.9 Data flow equations of an if statement ..................................... 33

2.10 Data flow equations for Iterative statements ................................ 34

2.11 Program dependence graph of the sum program in Figure 2.1. Bold arrows correspond to control flow dependencies, whereas the thin arrows shows data flow dependencies. .................................................. 35

2.12 A program $P$ and its end-slice with respect to the variable $\text{sum}$ .................. 37

2.13 Dynamic slice of the program in Figure 2.12 .............................. 39
2.14 Program $P$ and its backward and forward slices w.r.t. $(y, 2)$. ........................ 42
2.15 A multi-procedure program $P$. ................................................................. 43
2.16 Slice of the program in Figure 2.15 w.r.t. $z$ in procedure $\text{Increment}$. .... 45
2.17 Hausler’s algorithm for computing needed variables—$N_h$. ...................... 49
2.18 Hausler’s denotational slicing algorithm. .................................................. 50
2.19 A single-procedure program $P_2$. .......................................................... 51
2.20 Program dependence graph (PDG) of the program in Figure 2.19. ............ 52
2.21 PDG of the slice of the program in Figure 2.19, w.r.t. the variable $\text{prod}$.
   Bold arrows correspond to Control-flow dependences, whereas thin arrows
   correspond to data-flow dependences. ......................................................... 53
2.22 A multi-procedural program $P$. This program has been taken from $[50]$ .... 55
2.23 System dependence graph (SDG) of the program in Figure 2.22. Boldface
   edges represent control dependences, arcs represent data-flow dependences,
   where dashed lines represent $\text{Parameter-in}$, $\text{Parameter-out}$ and call edges. 56
2.24 Slice of SDG in Figure 2.23 resulting after the first phase. ....................... 58
2.25 Slice of SDG in Figure 2.23 as resulting after the second phase. ............... 59
2.26 Slice of SDG in Figure 2.23 as resulting after the first and the second phase. 60
2.27 Standard denotational semantics. .............................................................. 68
2.28 Weakest Precondition Semantics. ............................................................ 69
2.29 Strongest Postcondition Semantics. .......................................................... 71
2.30 A non-terminating program $P$. ............................................................... 73
2.31 Notation used in static meaning function. ............................................... 77
2.32 $\textit{Static meaning}$ function. ................................................................. 78
2.33 Interpretation function $\mathcal{I}$. ............................................................. 79
2.34 $P_1$ and $P_2$ do not have to same lazy semantics w.r.t. $x$. ....................... 80
2.35 Program in which the transfinite meaning of $y$ is $\omega$ .......................... 81
2.36 $P_1$ and $P_2$ do not have the same transfinite semantics w.r.t. $x$. ............ 82
2.37 Structural operational semantics for while language. ............................... 85

3.1 Needed variables (data dependence) .................................................. 88
3.2 Needed variables (control dependence) ............................................... 89
3.3 Needed variables (data and control dependence) .................................. 89
3.4 Hausler needed is not the same as Semantically Needed ......................... 89
3.5 $\mathbb{N}(P; Q, \{x\}) \subset \mathbb{N}(P; \mathbb{N}(Q, \{x\}))$ .................. 95
3.6 A non-terminating program $P_{3.6}$ and its slice w.r.t. the variable $x$. .... 100

4.1 Referenced and det of an expression. .............................................. 103
4.2 Recovering the value of $x$. ......................................................... 104
4.3 With initial state $\{x \mapsto 1, y \mapsto 1, z \mapsto \bot\}$, $x$ has 1 as its final value, whereas
\hspace{1cm} $y$ has $\bot$. ................................................................. 104
4.4 The lazy value of $x$ is 1 and of $y$ is $\bot$. .................................. 107
4.5 $x$ is evaluated to 1 where the final value of $z$ is $\bot$. ....................... 108
4.6 $P_1$ and $P_2$ have the same lazy semantics w.r.t. the variable $x$. ......... 115
4.7 $P$ and its end-slice have the same lazy semantics w.r.t. $x$. ............... 116
4.8 A simple program $P$. .............................................................. 121
4.9 $P_1$ and $P_2$ have the same lazy semantics w.r.t. the variable $x$. ...... 122
4.10 A simple program $P$ with an infinite loop. .................................... 123

5.1 A simple program $P$ with $\mathbb{N}(P, \bot) = \{y\}$. ......................... 135

6.1 Program $P_1$ and its corresponding slice w.r.t $z$ ............................ 147
6.2 Auxiliary functions ................................................................. 148
6.3 Interprocedural Slicing of Expressions ........................................ 151
6.4 Interprocedural Slicing of Statements ........................................ 154

7.1 A simple C program $P$ and its corresponding slices w.r.t. $x$ and $y$. .... 161
7.2 A program $P'$ equivalent to $P$ in Figure 7.1 and its corresponding slices
w.r.t. $x$ and $y$. ................................................................. 162
7.3 The program language under consideration. ................................. 163
7.4 A simple program $P$. ......................................................... 164
7.5 Unfoldings of the procedure $g$ in Figure 7.4. ............................... 164
7.6 Unfoldings of the procedure $p$ in Figure 7.4. ............................... 164
7.7 A program $P$ with two nested loops. ...................................... 167
7.8 An $P$-equivalent, $P'$, with two recursive functions. ..................... 168
Chapter 1

Introduction

Traditionally, slicing algorithms and program analysis algorithms in general are defined in terms of a variety of intermediate graph representations: control flow graphs, definition-use chains [2], data dependence graphs [60], program dependence graphs [26], program dependence web [4], program representation graphs [15] and static single assignment forms [19].

There have been many efforts to give a formal semantics of these intermediate program representations [15, 47]. Horwitz et al. [47] have shown that two programs with the same program dependence graph have the same semantics. Cartwright and Felleisen [15] defined a non-strict semantics for program dependence graphs of a simple procedural language. Other efforts give a construction definition of the program dependence graph by transforming the denotational semantics of imperative languages. The fact that these semantics are defined for some intermediate graphs representations instead of the programming language itself makes it difficult to model (prove correctness) program manipulation techniques such as slicing.

Program slicing [85] produces simpler programs from complicated ones and so can be thought of as a form of program transformation. Traditional program slicing [85, 88] is a technique for isolating the components of a program which are concerned with the computation of a single variable or a set of variables at some point in the program. Slices are constructed with respect to a slicing criterion, \((V, n)\), for some set of variables \(V\) and a program point \(n\). One reason that correctness proofs of traditional slicing algorithms are
so difficult may be that before we can start, a semantics of the intermediate structure is
required as well as mappings between programs and these intermediate structures in both
directions.

One aim of this thesis is to investigate slicing without intermediate structures. We regard
the intermediate structures as mere ‘implementation details’. Everything, the slicing algo-

rithm and the semantics of the program language and the ‘correctness criteria’ of slicing
are now expressed denotationally. This allows the possibility of using the full power and
elegance of denotational semantics in definitions and correctness proofs.

In 1989, Hausler [43] introduced the idea of expressing a slicing algorithm denotionally.
He defined an end-slicing\(^1\) algorithm using a number of denotational rules for a simple im-
perative language consisting of assignments, statement sequences, conditionals, and loops
but no subroutines. He did not prove his algorithm correct and no correctness proof can
be found in the literature. This, naturally, became our initial goal.

In order to prove his algorithm, a satisfactory definition of correctness had to be given.
To be useful, this definition had to be denotational. According to Weiser [88], a program
and its (end) slice must agree with respect to the set of variables in the slicing criterion.
In other words, if we run the original program and the slice, then, in all states where
the original terminates, the slice must also terminate with the same final values for the
variables in the slicing criterion. This is the correctness criterion that needs to be proved
for any slicing algorithm. The behaviour of the slice in states where the original does
not terminate is left undefined. In fact, traditional slicing algorithms sometimes introduce
termination: the standard semantics of a program is thus, less defined than the semantics
of some of its slices. Because of this it is unnatural to try to prove correctness of slicing
properties using the standard semantics [33]. This was observed and first discussed by
Cartwright and Felleisen [15].

\(^1\)the point of interest in the slicing criterion is the end-program point
In an attempt to solve this problem, Cartwright and Felleisen [15] introduce a non-strict lazy semantics of program dependence graphs. In lazy semantics, the standard semantics is made non-strict by making the update of the store function non-strict. As a result of this undefined values produced as a result of irrelevant computation are ignored: an expression is never computed until its value is required from the store. For example, in the program $P$ in Figure 1.1, the value of the variable $y$ after executing the program $P$ is $\perp$, however, its computation is irrelevant to the computation of the variable $x$. Therefore, the value of the variable $x$ using Cartwright and Felleisen [15] semantics is just 5.

```
x:=1;
while (x>0) y:=y+1;
x:=5;
```

Figure 1.1: A simple program $P$

This semantics loses precision for all variables defined in the body of if or while statement, i.e. they are evaluated to $\perp$ in states where their corresponding predicate is evaluated to $\perp$. This is due to fact that the evaluation of any expression requires its controlling predicate to be evaluated first. As a result of this, the lazy semantics of Cartwright and Felleisen [15] is not substitutive, i.e. replacing a subprogram $Q$ of a program $P$ with another semantically equivalent subprogram, $Q'$, does not guarantee that the resulting program $P'$ is semantically equivalent to the original program $P$. This lack of substitutivity means that their semantics is not really useful for proving correctness of the sorts of denotational transformations, described in this thesis, where a program’s transformation is defined in terms of transformations of its sub-components.

Giacobazzi and Mastroeni [33] argued that if a semantics is to be useful for modelling kinds of program manipulation such as slicing it should be able to capture semantic information ‘beyond infinite loops’ and be compositional. They do not consider the stan-
standard definition of compositionality of the semantics, where the semantics of the program is defined in terms of the semantics of its sub-program. Their definition of compositionality is restricted to a sequence of statements only. We call this property \textit{sequentiality} of the semantics instead of compositionality, to avoid any confusion with the standard definition of compositionality. They use transfinite states traces of programs \cite{55} and show the existence of such semantics using domain equations. They introduce a non-standard semantics, called \textit{transfinite semantics} using a metric structure on their value domains. \textit{Transfinite semantics} of a program is defined in terms of the set of all \textit{possibly transfinite computations}: computations whose length can be any ordinal, finite or infinite.

\begin{verbatim}
  x:=1;
  y:=1;
  while (y>0)
  {
    y:=y+1;
  }
\end{verbatim}

Figure 1.2: Transfinite meaning of $y$ is $\omega$

For example, the semantics traces in transfinite semantics of the program in Figure 1.2 is given as follows:

$((x, 1), (y, 1)) \rightarrow ((x, 1), (y, 2)) \rightarrow ((x, 1), (y, 3)) \rightarrow \cdots \rightarrow ((x, 1), (y, n)) \rightarrow \cdots ((x, 1), (y, \omega))$

Where $\omega$ is the first infinite ordinal. After executing the first infinite loop the variable the variable $y$ has the value $\omega$.

However, in this semantic, just in the case of the semantics of Cartwright \cite{15}, if an assignment $x:=e$ is controlled by an undefined predicate, then the variable $x$ is evaluated to $\bot$, i.e. if a variable $x$ is defined in the body of a while loop, in states where the while predicate is evaluated to $\bot$ the final value of $x$ becomes $\bot$ as well. Because of this, this
Figure 1.3: $P_1$ and $P_2$ do not have the same transfinite semantics w.r.t. $x$.

semantics is not compositional in the standard sense and therefore is not substitutive. For example, assignment $x:=x$ is semantically equivalent to skip statement with respect to the transfinite semantics. However, the programs $P_1$ and $P_2$ given in Figure 1.3 are not equivalent with respect to this semantics, i.e the final value of the variable $x$ when executing $P_1$ is $\perp$ and when executing $P_2$ the final value of $x$ becomes 1.

Central to slicing is the concept of variable dependence (or neededness as we call it): the set of variables needed by a set of variables $V$ in program $P$, noted as $N(P,V)$. Intuitively, this is the set of variables whose initial value ‘may affect’ the final value of at least one variable $v$ in $V$ after executing $P$. Our aim is to make the phrase ‘may affect’ semantically precise.

Neededness should be semantically discriminating. This corresponds to our intuitive understanding of neededness. i.e. if $x$ and $y$ are variables such that there exists two initial states $\sigma_1$ and $\sigma_2$, differing only on $x$, such that the meaning of $P$ gives rise to final terminating states with different values of $y$. Then $y$ should be needed by $x$ with respect to
A definition of neededness, clearly will also have to be consistent with Weiser’s algorithm [88]. That is, all variables that are ‘semantically needed’ must also be ‘Weiser Needed’. Otherwise, Weiser’s algorithm would be deemed in some cases not to produce valid slices. We cannot, however, expect the converse. Since Weiser’s algorithm works on data-flow equivalence classes of programs there will always be examples where Weiser-needed variables are not semantically needed. We define $y$ to be Weiser-Needed by a set of variables, $V$, in program $P$ (written $y \in N_W(P, V)$), if and only if slicing $y:=z; P$ with respect to $V$ using Weiser’s Algorithm gives $y:=z; Q$ where $Q$ is the slice of $P$ with respect to $V$ obtained by Weiser’s algorithm.

Finally we require neededness to be sub-sequential, in the sense that:

$$N(P; Q, V) \subseteq N(P, N(Q, V)).$$

If this was not the case, then it would mean that there was a variable $z$ which affects the value of $x$ in $P; Q$ but for no variable, $k$, which affects the value of $x$ in $Q$, does $z$ affect the value of $k$ in $P$.

Standard semantics loses precision in the presence of infinite loops. Due to issues regarding non-termination, it turns out to be hard, if it is not impossible, to define neededness in terms of the standard semantics. Much of our research was trying to find a semantics which allowed us to define neededness satisfactorily that was consistent with standard semantics. For this semantics to be useful for program slicing, it has to be substitutive.

This leads to a new lazy semantics defined in Section 4.1, Chapter 4 which is at the heart of our work. Unlike the semantics of Cartwright and Felleisen [15], our semantics does not use any intermediate graph representation. It is defined denotationally for each construct of a simple while language. In addition it is proved that it is substitutive. Furthermore, in terms of our lazy semantics the definition of neededness turns out to be straightforward.
In Section 4.1 we give a new semantic definition of a slice which is consistent with Weiser’s [88] definition. That is a slice has to preserve both termination and the lazy semantics of the original program with respect to slicing criterion.

As a demonstration of the applicability of our lazy semantics, in Chapter 5, Hausler’s Denotational Slicing Algorithm is proved correct with respect to the lazy semantic definition of a slice given in Section 4.1. Since our lazy definition of a slice is stronger than the standard one, this proves that Hausler’s Algorithm [43] is correct with respect to the standard definition too.

In Chapter 7 a summary of our contributions and a broad discussion of the future direction of our work are given. In Section 7.1 an overview of the contribution of this thesis is given. As a direction of our future work we will explore the extension of our lazy semantics and Hausler’s slicing algorithm to handle programs with recursive procedures. To do this, a number of issues have arisen due to recursion. These issues are broadly discussed in Section 7.2.
Chapter 2

Background

Most program slicing algorithms [88, 73, 50] are defined in terms of a variety of intermediate graph representations: control-flow graphs, definition-use chains [2], data-dependence graphs [60], program-dependence graphs [26], program-dependence web graphs [4], program-representation graphs [15] and static-single assignment forms [19].

2.1 Control Flow

In high-level languages, control structures express the flow of control. For example the boolean expression of an if decides which branch will be executed. These control structures can be translated into conditional and unconditional jumps in low-level languages.

2.1.1 Control Flow Graphs

Control-flow graphs [2] are an abstract representation of programs. They represent all possible sequences of statements in a program's execution. They have been used as a basis for data-flow analysis [68].
2.1 Control Flow

Definition 1 (Control-Flow Graph): A control-flow graph of a program \( P \) is a directed graph in which each node is associated with a statement from \( P \). Two additional nodes, Entry and Exit, correspond to the beginning and the end of the program \( P \). With each node \( n \) associate two sets: \( \text{Ref}(n) \), the set of variables referenced at \( n \), and \( \text{Def}(n) \), the set of variable defined at \( n \). The edges edges represent the flow of control in \( P \). An edge between two nodes \( n \) and \( m \), denoted \( n \rightarrow m \), represents a control flow from \( n \) to \( m \).

For each node, \( n \), in the control-flow graph there exists a path from Entry to Exit containing \( n \). Figure 2.2 shows the control-flow graph of the program in Figure 2.1.

```java
i:=1;
sum:=0;
while (i<11)
{
    sum:=sum+i;
    i:=i+1;
}
write(sum);
```

Figure 2.1: Sum program \( P \).

2.1.2 Post-dominance in the Control-Flow Graph

Definition 2: A node \( u \) post-dominates a node \( v \), if and only if all paths from \( v \) to exit pass through \( u \).

Definition 3: We call a node \( u \) post-dominator of a node \( v \), if and only if all paths from \( v \) to exit pass through \( u \).
Figure 2.2: CFG of the program in Figure 2.1 where Def and Ref are the set of variables defined and referenced at a program point.
2.1 Control Flow

Definition 4: We call \( u \) the immediate post-dominator of a node \( v \), if and only if \( u \) is a post-dominator of \( v \), \( u \neq v \), and there is no other node \( w \), for which \( u \) is a post-dominator and that it is itself a post-dominator of \( v \).

2.1.3 Control Dependences

The notion of control dependence was first introduced by Ferrante [26].

Definition 5: Let \( G \) be a control-flow graph. Let \( u \) and \( v \) be nodes in \( G \). We say \( u \) is control dependent on \( v \) if and only if all of the following hold.

1. There exists a directed path \( \pi \) from \( v \) to \( u \).

2. For all nodes \( w \) in \( \pi \) with \( u \neq w \neq v \), \( w \) is post-dominated by \( u \).

3. \( v \) does not post-dominate \( u \).

If a node \( u \) is control-dependent on \( v \) then \( v \) must have multiple successors, where \( u \) is executed in one and only one path from \( v \).

Definition 6: The Control-dependence graph over the control-flow graph \( G \) is the graph over all nodes of \( G \) in which there is a directed edge from node \( u \) to node \( v \) if and only if \( v \) is control dependent on \( u \).

In the control-dependence graph there is an edge from each predicate to every statement controlled by it. No information about data dependences is included. In control dependence graphs edges are marked True or False. For example there is a True edge from a predicate of an if statement to each statement in the True branch and a False edge to each statement in the False branch.

The direction of the dependence indicates the flow of control. Figure 2.3 shows the control dependences of the program shown in Figure 2.1.
while(i<11) {sum = 0; i = 1; write(sum); i = i+1; sum = sum+i; while(i<11); write(sum); i = i+1}

Figure 2.3: Control dependencies of the program in Figure 2.1
2.2 Data Dependence

Data-dependences give information about the flow of a variable’s value from the points of variable definitions to the points of their use. If a value computed at a node $v$ is dependent on the value computed at a node $u$, then $v$ is data-dependent on $u$.

**Definition 7**: A data-dependence graph is a graph representation over the abstract syntax tree of a program. A data dependence edge from a node $u$ to node $v$ exists if and only if

- there exists a variable $x$, defined in $u$ which is referenced in $v$ and,
- there exists an execution path $\pi$ reaching $v$ after $u$ without a redefinition of $x$.

We also say the node $v$ is data-dependent on node $u$. This means that the definition of a variable at the node $u$ reaches the node $v$. We say that the node $u$ is a reaching definition for $v$. Data flow analysis is the computation of reaching definitions [68].

Figures 2.4 shows the data dependence graph of the simple program shown in Figure 2.1.

2.2.1 Computation of Used(Referenced) and Defined Variables

In order to compute the Reaching Definitions, we need to compute the set of variables that are used(referenced) by each statement and the set of variables that are defined in each statement.

From the graph representation of the program, the set of variables used by a statement of the dependence graph is computed easily. Figure 2.5 shows used and defined variables for an assignment statement.

When a variable $x$ is assigned a new value, the old value is killed and is replaced by the
while(i<11) 
write(sum) 
i = i+1 
sum = sum+i 
i = 1 
sum = 0

Figure 2.4: Data-flow dependencies of the program in Figure 2.1
2.2 Data Dependence

<table>
<thead>
<tr>
<th>Statement</th>
<th>Used</th>
<th>Defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := y + z$</td>
<td>$y, z$</td>
<td>$x$</td>
</tr>
</tbody>
</table>

Figure 2.5: Used and defined variables

new one. This is a killing definition of the variable $x$. This is shown in Figure 2.6.

| $x := 0$; | definition of the variable x |
| $y := 1$; | definition of the variable y |
| $x := 2$; | new definition of $x$ kills the first one |
| $v := x$; | only the new definition of $x$ is reaching $v$ |

Figure 2.6: Example illustrating $gen$ and $kill$

2.2.2 Computation of Reaching Definitions

After computing the sets of used and defined variables for every single statement, the reaching definition can be computed. To identify every definition, labels are given to each one of them. Five types of sets are used, definition set, $gen$ set, $kill$ set, $in$ set and $out$ set. In the following we define each one of them:

**Definition 8**: The definitions of the variable $x$, definitions($x$), represents the set of all labels that define $x$.

**Definition 9**: The $gen$ set of a statement $S$ represents the set of all labels of all definitions that are generated by $S$.

**Definition 10**: The $kill$ set of a statement $S$, kill($S$), represents the set of all labels of all definitions that are killed by $S$.

**Definition 11**: The $in$ set of a statement $S$, in($S$), represents the set of all labels of all definitions that reach $S$. 
2.2 Data Dependence

Definition 12: The out set of a statement $S$, $out(S)$, represents the set of all labels of all definitions that leave $S$.

To compute the $gen(S)$, and the $kill(S)$ sets and the reaching definitions in a program $S$, we need to solve data-flow equations for all statements of the program $S$.

Data Flow Equations for Assignments

Figure 2.7 shows data-flow equations of an example of a killing assignment $S$. ($l$) represents the label of the assignment $S$, which is the only element of the $gen$ set as the statement $S$ generates only the definition $l$ for the variable $x$. In addition to that the statement $S$ kills all other old definitions of the variable $x$. The $out$ set will contain all of the definitions that are generated by $S(gen(S))$ and the definitions from the $in(S)$ set which are not killed by the statement $S$.

\[ \text{out} = \text{gen}(S) \cup (\text{in}(S) - \text{kill}(S)) \]

Figure 2.7: Data-flow equations for an assignment
Data Flow Equations for sequences—$S_1; S_2$

Figure 2.8 shows data flow equations for a sequence of statements $S_1$ and $S_2$. The figure shows that the effect of the statement sequence is affected by the effect of $S_1$ and $S_2$. The union of the set $gen(S_2)$ (all definitions generated by $S_2$), and the definitions generated by $S_1$ which are not killed by $S_2$, represents the set of all definitions generated by the compound statement ($gen(S_1; S_2)$). $kill(S_1; S_2)$ will represent the union of all definitions killed by the statement $S_2$ ($kill(S_2)$), and the set of all definitions killed by $S_1$ which are not generated by the statement $S_2$ ($kill(S_1) - gen(S_2)$). $out(S_1; S_2)$ is the union of $in(S_1; S_2) = in(S_1)$ and all the definitions generated by $S_1; S_2$ minus all the definitions killed by the compound statement. Hence, after substitution we get:

$$out = gen(S_2) \cup (gen(S_1) - kill(S_2)) \cup (in(S_1) - (kill(S_2) - (kill(S_1) - gen(S_2))))$$

Data Flow Equations for selective statements

Definitions that are generated by selective statements such as conditionals are the union of definitions generated by each of its branches. A definition of a selective statement is killed if and only if it is killed by all branches. Otherwise, it is unknown which branch will be executed and hence, it is not killed. Figure 2.9 gives data-flow equations for a selective statement (selection of the $\text{then}$ and $\text{else}$ branches of an $\text{if}$ statement).

Data Flow Equations for iterative statements

The $gen$ and $kill$ sets of an iterative statement such as $\text{while}$ loops are the same as the ones for a nested statement sequence. A definition generated after the first iteration is also generated after any iteration. Figure 2.10 shows data flow equations for an iterative
\[ \text{in} = \text{in}(S2) = \text{out}(S1) \]

\[ \text{out}(S1) = \text{in}(S2) \]

\[ S = S1;S2 \]

- \( \text{gen} = \text{gen}(S2) \cup (\text{gen}(S1) - \text{kill}(S2)) \)
- \( \text{kill} = \text{kill}(S2) \cup (\text{kill}(S1) - \text{gen}(S2)) \)
- \( \text{in}(S1) = \text{in} \)
- \( \text{in}(S2) = \text{out}(S1) \)
- \( \text{out} = (\text{gen} \cup \text{in}) - \text{kill} \)

Figure 2.8: Data-flow equations for sequence of statements—\((S1;S2)\)
\begin{center}
\begin{tikzpicture}
    \node (S1) at (0,0) {S1};
    \node (S2) at (1,0) {S2};
    \node (in) at (-0.5,1) {in};
    \node (out) at (0.5,1) {out};
    \draw[->] (in) -- (S1);
    \draw[->] (S1) -- (S2);
    \draw[->] (S2) -- (out);
    \draw[->] (in) -- (out);
\end{tikzpicture}
\end{center}

\begin{itemize}
    \item gen = gen(S1) \cup gen(S2)
    \item kill = killt(S1) \cap kill(S2)
    \item in(S1) = in(S2) = in
    \item out = out(S1) \cup out(S2)
\end{itemize}

Figure 2.9: Data flow equations of an if statement.
statement.

![Diagram](image)

```
gen = gen(S)
kill = kill(S)
in(S) = in U out(S)
out = gen(S) U in U out(S) - kill(S)
```

Figure 2.10: Data flow equations for Iterative statements.

The computation of data-flow equations of a statement $S$ can be done in two steps. We first compute the $gen$ and $kill$ sets for each statement of the program then for each statement the $out$ set is computed as from the $gen$, $kill$ and $in$ sets as follows:

$$out(S) = (gen(S) \cup in(S)) - kill(S).$$

### 2.3 Program dependence graph

A program dependence graph [59, 26] is a program graph representation, which combines both control dependences and data dependences. Statements and predicate expressions represent the nodes on the program dependence graph. The edges incident to a node represent both the data value on which the node’s computation depends and the control
condition on which the execution of the node’s computation depends. Figure 2.11 shows the program dependence graph of the program in Figure 2.1.

![Program dependence graph](image)

Figure 2.11: Program dependence graph of the sum program in Figure 2.1. Bold arrows correspond to control flow dependencies, whereas the thin arrows show data flow dependencies.

Program dependence graphs remove the sequencing constraints of the program text and explicate the inherent parallelism of a program. The graph representation also facilitates the manipulation of programs and simplifies the program and graph transformations required by, for example, optimising compilers. These transformations become just creation and deletion of nodes and edges. Ottenstein and Ottenstein [73] used the program dependence graph to define slicing. Program dependence graphs capture both control dependences and data dependences information and therefore the behaviour of the program is captured as well. Horwitz et al [47] show that the program dependence graph does capture the program’s behaviour. Different semantics of program dependence graph have been developed
depending on the intended application [47, 15]. These semantics are given later on (see Section 2.7).

2.4 Program Slicing

Program slicing [85, 86, 87] is a technique to decompose programs based on the analysis of the control and data flow. It reduces a program to those statements that are relevant for a particular sub-computation. The original definition of a program slice was introduced in 1979 by Mark Weiser [85]. Weiser claims that a slice corresponds to the mental abstractions that programmers make when they are debugging a program, and suggests the integration of program slicers in debugging environments. Weiser’s definition of program slicing is based on statement deletion. A slice of a program P consists of any subset of statements of P preserving the behaviour of the original program with respect to a program point p and a subset of program variables V, referred to as a slicing criterion, \( (V, p) \). Program slicing has many applications including reverse engineering [14, 78], program comprehension [23, 41], software maintenance [13, 17, 31, 32], debugging [1, 53, 63, 89], testing [8, 35, 37, 45, 46], component re-use [5, 16], program integration [11, 48], and software metrics [7, 61, 72]. There are several surveys of slicing techniques, applications and variations [10, 22, 39, 80].

2.4.1 Different Types of Program Slicing

In this section we describe different types of program slicing.

Static slicing

Static slicing [88] uses static analysis to derive slices, i.e. the source code of the program is analysed and the slices are computed for all possible input values, i.e., no assumption is
made about the input values. Weiser [88] defined the slicing criterion to be a pair \((V, N)\), where \(V\) is subset of program variables and \(N\) is a program point. A static slice of a program \(P\), with respect to a slicing criterion \((V, N)\), is a sub-program of \(P\) consisting of all statements that may affect the values of the variables in \(V\) at a program point \(N\) for every possible execution, i.e., the slice must preserve the behaviour of the original program at the program point \(N\) with respect to the variables in \(V\).

A static end-slice of program \(P\) with respect to a set of variables \(V\) is any sub-program of \(P\) which preserves the projection of the semantics of the original program with respect to \(V\) for every possible execution of the program. It is just a static slice where the program point of interest in the slicing criterion is just the end program point. Figure 2.12 shows a simple program \(P\) and its static end-slice with respect to the variable \texttt{sum}. Weiser used a control flow graph as an intermediate representation for his static slicing algorithm. Since Weiser’s original proposal, different algorithms for computing static slices [73, 50] have been developed using different intermediate graph representations such as program dependence graphs [26, 73] and system dependence graphs [50]

```
sum:=0;
prod:=1;
while (i<=10)
{  sum:=sum+i;
    prod:=prod*i;
    i:=i+1;
}
printf(sum);
printf(prod);
```

```
sum:=0;
while (i<=10)
{  sum:=sum+i;
    i:=i+1;
}
printf(sum);
```

A simple program \(P\)  
Static end-slice w.r.t. \texttt{sum}.

Figure 2.12: A program \(P\) and its end-slice with respect to the variable \texttt{sum}. 
Dynamic program slicing

*Dynamic slicing* was first introduced by Korel and Laski [57]. Dynamic slicing makes use of the information about a particular execution of a program. Unlike static slicing, where the slice has to preserve the behaviour the original program upon the slicing criterion in all possible path executions, a dynamic slice has to preserve the effect of the original program upon the slicing criterion upon a particular path execution only. This execution is characterised by a particular initial program state.

In dynamic slicing, the traditional static slicing criterion is augmented with an initial input program state. Dynamic program slicing isolates the program components that may affect the final values of the variables of interest for the given input. During debugging, programmers generally analyse the program behaviour under the test-case that revealed the error, i.e. a bug in a program is detected as the result of the execution of the program with respect to some specific input.

Dynamic slicing [57] was originally introduced for program debugging [1, 58] as it produces more accurate and smaller slices in size than those produced by using traditional static slicing [88]. The application of dynamic slicing has been extended well beyond debugging to other program analysis techniques such as program comprehension, software testing and software maintenance [25, 36, 53].

Figure 2.13 shows the dynamic end-slice of the program $P$ in Figure 2.12 with respect to variable $sum$ for the input $i=11$.

Quasi Static Slicing

*Quasi-static slicing* was first introduced by Venkatesh [81] as a bridge between the two extreme forms of slicing, static and dynamic. A quasi-static slice has to preserve the behaviour of the original program with respect to the set of variables of interest on a
subset of program inputs. In some applications, such as testing, it is appropriate for the value of some input variables to be fixed while the value of the other vary. This was the main motivation of quasi-static slicing.

**Conditioned Slicing**

Canfora et al [12] were the first to introduce **conditioned slicing**. **Conditioned slicing** [12] is a general framework for statement deletion based slicing. **Conditioned slicing** forms a theoretical bridge between the two extremes of static and dynamic slicing. A conditioned slice consists of a subset of program statements which preserves the behaviour of the original program with respect to a slicing criterion for a given set of program executions. These executions are characterised by some set of initial program states. This set is captured, in conditioned slicing, by augmenting the traditional slicing algorithm with a condition on the input. Hence, the slicing criterion in conditioned slicing becomes $\langle V, n, c \rangle$, where $V$ is the set of variables of interest, $n$ is program point and $c$ is a condition on the program input. The condition $c$ can be imposed at any point, $m$, of the program. In this case the slicing criterion becomes $\langle V, n, c, m \rangle$. This additional condition can be used to simplify the program before applying a traditional static slicing algorithm. Such pre-simplification is called conditioning and it is achieved by eliminating statements which do not contribute to the computation of the variable of interest when the program is executed in an initial
state which satisfies the initial condition in the slicing criterion.

A conditioned slice of program $P$ with respect to slicing criterion, $⟨V, n, c⟩$ can be achieved by first simplifying the program with respect to the condition $c$ on the input (i.e. discarding infeasible paths with respect to the condition $c$) and then computing the slice on the reduced program with respect to $⟨V, n⟩$. A symbolic executor [56, 18] can be used to compute the reduced program, also called conditioned program in [13]. A conditioned slice of a program $P$ with respect to the slicing criterion $⟨V, n, c⟩$ is any sub-program of $P$ which preserves the projection of the semantics of $P$ with respect to $⟨V, n⟩$ in all initial program states satisfying the condition $c$. Danicic et al [29] introduced an implementation of conditioned slicing for an intraprocedural subset of $C$ programming language. They introduced ConSIT, which is the first fully automated implementation of conditioned slicing. ConSIT incorporates conventional static slicing, symbolic execution and theorem proving.

Conditioned slicing allows a better decomposition of the program giving human readers the possibility to analyse code fragments with respect to different perspectives. Canfora et al. [12] have demonstrated that conditioned slicing subsumes any other form of statement deletion based slicing method, i.e., the conditioned slicing criterion can be specified to obtain any form of slice.

Different variants of conditioned slicing have been presented in the literature [27, 70]. Ning et al. [70] proposed a tool called COBOL/SRE, to extract different types of slice from legacy systems, in particular condition-based slices. The user specifies a logical expression and a slicing range and the tool automatically isolates the statements that can be reached along control flow paths under the given condition. However, the authors did not propose a formal definition of conditioned based slicing. Field et al. [27] introduced the concept of constrained slice to indicate slices that can be computed with respect to any set of constraints. Their approach is based on an intermediate representation of imperative programs, named PIM, and exploits graph rewriting techniques based on dynamic dependence tracking [28]. The slices extracted are not executable. The authors are more
interested in the semantics than in simple statement deletion.

An extension of conditioned slicing, namely backward conditioning, has been proposed by Danicic et al. [30]. While conditioned slicing uses forward conditioning and deletes statements that are not executed when the initial state satisfies the condition, backward conditioning deletes statements which cannot cause execution to enter a state which satisfies the condition. Backward conditioning addresses questions of the form “what parts of the program could potentially lead to the program arriving in state satisfying a given condition”, whereas forward conditioning addresses questions of the form “what happens if the program starts in a state satisfying a given condition?”

Conditioned slicing has been applied to program comprehension [30, 23] and to the extraction of reusable functions [13, 70].

Amorphous Slicing

Amorphous slicing was first introduced by Harman and Danicic [38]. Amorphous slicing [9, 38, 42] is a combination of program transformations and traditional slicing. Unlike traditional slicing, where the syntax of the original program is preserved, program transformations can be used to reduce the size of the program where a projection of the semantics is preserved. Amorphous slicing is a variation of program slicing where the syntax of the program does not have to be preserved but the semantic is preserved with respect to the slicing criterion. The fact that amorphous slicing combines both transformations and traditional leads to the construction of more refined slices, than those produced using traditional slicing on its own. Amorphous slicing is more useful if the size of a slice is more important than the preservation of syntax. To prove correctness of this type of transformations where the syntax of the program is necessarily preserved, requires the semantics to be substitutive.
2.4 Program Slicing

![Image of program slicing]

Figure 2.14: Program $P$ and its backward and forward slices w.r.t. $(y, 2)$.

**Backward and Forward Slicing**

Program slices, as originally introduced by Weiser [88], are now called backward slices, because they contain all parts of the program that might have influenced the variable at the statement under consideration. On the other hand, forward slices contain all parts of the program that could be influenced by the variable of interest. Figure 2.14 shows a program $P$ and its corresponding backward and forward slices with respect to the variable $y$ at the program point 2.

**Intraprocedural and Interprocedural Slicing**

*Intraprocedural slicing* computes slices within a procedure. Calls to other procedures are either not handled at all or handled conservatively. If the program consists of more than one procedure, *interprocedural slicing* can be used to derive slices that span multiple procedures.

Interprocedural slicing raises a new problem: when a procedure is called at different places, the calling context must be considered, in order to correctly model the run-time execution at compile time. Interprocedural data flow analysis has a similar goal to only consider paths that correspond to legal call/return sequences. Such paths are called *realisable, valid or feasible*. Figure 2.15 shows a module where the procedure `Add` is called at two places:
once in the procedure \texttt{Increment}, another time in the procedure \texttt{A}.

\begin{figure}[h]
\centering
\begin{verbatim}
Add(a,b)
{ a:=a+b;
}

Increment(z)
{ Add(z,1);
}

A(x,y)
{ Add(x,y);
    Increment(y);
}

main()
{ sum:=0;
    i:=1;
    while (i<11)
        do A(sum,i);
}
\end{verbatim}
\caption{A multi-procedure program \textit{P}.}
\end{figure}

When computing the slice, "regarding the calling context" means that the slicing algorithm correctly models the execution. When the call of \texttt{Add} in the procedure \texttt{Increment} is encountered, it is necessary to continue the analysis with procedure \texttt{Add}. But when returning from procedure \texttt{Add}, analysis of the procedure \texttt{Increment} must be continued. It would not be a precise model of the run-time execution to call \texttt{Add} in the procedure \texttt{Increment} but to return to the procedure \texttt{A}.

If the procedure \texttt{Increment} is sliced for the output parameter \texttt{z} without regarding the calling context, the slice will contain the whole program. This is imprecise, if one allows
not only the deletion of entire statements from the original program but also of smaller parts such as individual parameters: the call of \texttt{Add} within procedure \texttt{A}, the first actual parameter of \texttt{A} at its call in the procedure \texttt{main} and the initialisation of \texttt{sum} are not relevant. The imprecision is introduced because \texttt{Add} is (necessarily) included into the slice (since it is called in \texttt{Increment}) and all call sites to \texttt{Add} are then (unnecessarily) included into the slice. The call to the procedure \texttt{Add} in \texttt{Increment} must be included into the slice, but the call to \texttt{Add} in the procedure \texttt{A} should not be included into the slice. The inclusion of the call to \texttt{Add} within \texttt{A} into the slice necessitates the inclusion of the formal parameter \texttt{x} of \texttt{A}, the corresponding actual parameter \texttt{sum} and the initialisation of \texttt{sum}. On the other hand, if the calling context is considered, the slice will only contain relevant parts, as shown in Figure 2.16. This is obtained by using slicing as a graph reachability problem over the system dependence graph by Horwitz [50]. A detailed description of slicing using system-dependence graph is given in the next section.

2.5 Current slicing algorithms

2.5.1 Weiser’s slicing algorithm

Program slicing was introduced by Weiser [85, 86, 87]. Weiser’s slicing algorithm consists of computing data flow information over the control flow graph of a program. He computes the set of variables relevant at each node of the control flow graph. Suppose there is an edge from a node \textit{i} to a node \textit{j}, then the set of variables relevant at node \textit{j} and the set of variables defined in \textit{i} will determine whether the node \textit{i} should be be left or deleted from the control-flow graph of the program to be sliced. The resulting control-flow graph represents the control graph of the program slice. Slices can be computed by solving a set of data and control flow equations derived directly from the control flow graph of the program being sliced. These equations are solved using an iterative process which computes the set of
Add(a,b)
{a:=a+b;}

Increment(z)
{Add(z,1);}

A(x,y)
{;Increment(y);}

main()
{;i:=1;
while (i<11)
do A(sum,i);
}

Figure 2.16: Slice of the program in Figure 2.15 w.r.t. z in procedure Increment.

relevant variables for each node in the control flow graph.

A Weiser-slice of a program with respect to $C = \langle n, V \rangle$ is defined by an algorithm that exploits the notion of directly and indirectly relevant referenced variables as follows:

**Directly relevant variables**

We first compute the set of directly relevant variables at each node of the control-flow graph of the program the be sliced. The directly relevant variables of a node $i$ with respect to the slicing criterion $C$, $R_C^d(i)$, are inductively defined as follows:
2.5 Current slicing algorithms

1. Initialise the relevant sets of all nodes in the control flow graph to the empty set.

2. The relevant set of variables at $n$ is just the set of variables of interest $V$.

3. The set of directly relevant variables at every other node $i$, is defined in terms of the set of directly relevant variables of all immediate predecessor nodes, $j$, of $i$: all nodes $j$ leading directly from $i$ to $j$ in the control flow graph. The relevant set of variables at a node $i$, $R_C^0(i)$, contains all the variables $v$, satisfying the following:
   
   \begin{itemize}
     \item $v \in R_C^0(j)$ and $v \notin Defined(i)$
     \item $v \in Referenced(i)$ and $Defined(i) \cap R_C^0(j) \neq \emptyset$.
   \end{itemize}

   The directly relevant variables of a node are the set of variables at that node upon which the slicing criterion is transitive data dependent.

**Directly relevant statement**

The set of directly relevant statements with respect to the slicing criterion $C$, $S_C^0$, is defined in terms of the directly relevant variables as follows:

$$S_C^0 = \{ i : \exists j \text{ such that } i \rightarrow_{CFG} j \text{ and } Defined(i) \cap R_C^0(j) \neq \emptyset \}.$$ 

Where $i \rightarrow_{CFG} j$ means there exists control edge from node $i$ to node $j$.

This completes the first iteration of Weiser’s algorithm.

**Indirectly relevant variables**

During the first phase, control-dependence is not taken into account. In order to do that, Weiser computes a new set of variables, indirectly relevant set of variables ($R_C^k$ where $k \geq 0$
and $C$ is the slicing criterion), for each node in the control flow graph. This set is now defined.

\[
R^{k+1}_C(i) = R^k_C(i) \cup \left( \bigcup_{b \in B^k_C} R^0_{\text{Referenced}(b),b}(i) \right).
\]

Where $B^k_C$ is the set of all predicate nodes that control a statement in $S^k_C$.

\[
B^k_C = \{ b : \exists i \in S^k_C \text{ such that } b \text{ controls } i \}.
\]

**Indirectly relevant statements**

Weiser’s slice is achieved by augmenting $S^k_C$ with the predicate nodes to include further indirectly relevant statements:

\[
S^{k+1}_C = B^k_C \cup \{ i : \exists j \text{ such that } i \rightarrow_{CFG} j \text{ and } \text{Defined}(i) \cap R^{k+1}_C(j) \neq \emptyset \}.
\]

Hausler [43] presents a denotational program slicer for a simple programming language without procedures. Hausler’s algorithm is just a denotational reformulation of Weiser’s algorithm. Hausler, unlike Weiser, does not use any intermediate graph representation.

In order to demonstrate the denotational approach introduced by Hausler [43], in Figure 2.18, we give his definition of an intraprocedural slicer:

\[
S_h : S \times \mathcal{P}(V) \rightarrow S.
\]

for a simple procedural programming language.
2.5 Current slicing algorithms

Where \[ \begin{align*}
S &= \text{set of programs,} \\
V &= \text{set of program variables}
\end{align*} \]

Hausler defines \( S_h \) denotationally. Two functions are required:

\( S_h(P, V) \), the slice of program \( P \) with respect to the set of variables \( V \) at the end program point, which takes a program \( P \) and a set of variables \( V \) and returns the resulting end-slice of \( P \) with respect to \( V \).

\( N_h(P, V) \), the Hausler-needed set with respect to the set of variables, \( V \), of program \( P \).

In Figures 2.18 and 2.17, we define \( S_h(P, V) \) and \( N_h(P, V) \) respectively in terms of the structure of the syntax of a simple procedural programming language.

One of the aims of this thesis is to prove correctness of Hausler’s slicing algorithms.

2.5.2 Slicing as a Graph-Reachability Problem

Slicing Program-Dependence Graphs

Ferrante et al. [26] introduced *program-dependence graphs* (PDGs) which combine control-dependences and data dependences into a common framework. The nodes of a program-dependence graph represent the statements and predicate expressions of a program. Each node of the graph has an edge both to the nodes that are control-dependent on it and also to the nodes that define its operands. The set of all dependences induce a partial ordering on the statements and predicates in the program that must be maintained in order to preserve the semantics of the original program.

They proposed the use of program dependence graphs for optimisation [26]. Since both the essential control relationships and the essential data relationships are present in the
program dependence graph, they noticed that a single traversal of these dependences is sufficient to perform many optimisation.

Ottenstein et al [73] were the first of many to define slicing as a graph reachability problem over the program-dependence graph. They use the program-dependence graph (PDG) for static slicing of a single-procedure programs. Figure 2.20 shows a program-dependence graph for a single-procedure program shown in Figure 2.19.

Figure 2.21 shows the PDG of the slice, with respect to the variable prod, of the program shown in Figure 2.19. It is extracted from the PDG of the whole program as result of
\( S_h : S \times P(V) \rightarrow S \)

skip statements
(1) \( S_h(\text{skip}, V) = \text{skip} \)

abort statements
(2) \( S_h(\text{abort}, V) = \text{skip} \)

Assignment statements
(3) \( S_h(x = e, V) = \begin{cases} \text{skip} & \text{if } x \notin V \\ x := e & \text{if } x \in V \end{cases} \)

Sequences of statements
(4) \( S_h(S_1; S_2, V) = S_h(S_1, \mathcal{N}_h(S_2, V)); S_h(S_2, V). \)

if statements
(5) \( S_h(\text{if } (B) S_1 \text{ else } S_2, V) = \begin{cases} \text{skip} & \text{if } S_h(S_i, V)_{i=1,2} = \text{skip} \\ \text{if } (B) S_h(S_1, V) \text{ else } S_h(S_2, V) & \text{otherwise.} \end{cases} \)

while loops
(6) \( S_h(\text{while } (B) S, V) = \begin{cases} \text{skip} & \text{if } S_h(S, V) = \text{skip} \\ \text{while } (B) S_h(S, \mathcal{N}_h(\text{while } (B) S, V)) & \text{otherwise} \end{cases} \)

Figure 2.18: Hausler’s denotational slicing algorithm.

graph reachability.

Interprocedural Slicing Using System Dependence Graphs

Horwitz et al [50] enhanced the program-dependence graph to facilitate interprocedural slicing. They added vertices for the program entry, for the initial definition of variables and for the final use of variables to the graph. They labelled control dependences with either True or False, meaning that when the predicate at the origin of the dependence evaluates to True (or False), the target node of the dependence which is labelled True (or False) will eventually be executed. They classify data dependences as loop carried


and loop independent. However, for slicing purposes the distinction between different kinds of data dependences is not necessary. For multi-procedure programs, Horwitz et al. [50] have defined the system dependence graph (SDG), which consists of a program dependence graph for each procedure of the program. They introduce several nodes to model procedure calls and parameter passing. Parameters are passed by value-result and accesses to global variables are modelled via additional parameters of the procedure.

- Call-site nodes represent procedure calls.

- Actual-in and Actual-out nodes represent the input and output parameters at the call-site node.

- Formal-in and Formal-out nodes represent the input and output parameters at the called procedure. They are also control dependency on the procedure’s entry node.
Figure 2.20: Program dependence graph (PDG) of the program in Figure 2.19.
2.5 Current slicing algorithms

Figure 2.21: PDG of the slice of the program in Figure 2.19, w.r.t. the variable prod. Bold arrows correspond to Control-flow dependences, whereas thin arrows correspond to data-flow dependences.
Horwitz et al also introduced additional edges to link the program dependence graphs together.

- **Call edges** link the call-site with the procedure entry node.
- **Parameter-in edges** link Actual-in nodes with Formal-in nodes.
- **Parameter-out edges** link the Formal-out nodes with the Actual-out nodes.

Finally, they introduced different kind of edges, called summary edges, to represent the transitive dependences due to the calls. A summary edge is added from an Actual-in node A to an Actual-out node B, if there exists a path of control, data and summary edges in the called procedure from the corresponding Formal-in A’ to the Formal-out B’. Summary edges can be used to simulate the effects of a call without descending into the sub-graph of the called procedure while still regarding the effect to the called procedure.

In order to compute the subordinate characteristic graphs, Horwitz et al. used a linkage grammar [54]. The linkage grammar is used only to compute transitive dependences due to procedure calls.

Figure 2.23 represents the system dependence graph for a multi-procedural program shown in Figure 2.22.

Interprocedural slicing can be implemented as a graph reachability problem over the system dependence graph. However, the transitive closure over all dependences yields a slice that does not take the calling context into account and therefore contains irrelevant nodes. To address this problem, Horwitz et al. [50] developed a two-phase algorithm that computes more precise interprocedural slices. Slices are taken with respect to a node \( n \) in the procedure \( P \). A brief outline of this algorithm is discussed in the following:

1. In the first phase, all edges except Parameter-out edges are followed backwards starting from the node \( n \) in the procedure \( P \). All nodes that either reach \( n \) and are in \( P \)
2.5 Current slicing algorithms

<table>
<thead>
<tr>
<th>proc A(x,y)</th>
<th>program Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Add(x,y);</td>
<td>i=1;</td>
</tr>
<tr>
<td>call Inc(y);</td>
<td>while(i&lt;11)</td>
</tr>
<tr>
<td>return</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>call A(sum,i);</td>
</tr>
<tr>
<td>proc Add(a,b)</td>
<td>}</td>
</tr>
<tr>
<td>a := a+b;</td>
<td>end(sum,i);</td>
</tr>
<tr>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

### Figure 2.22
A multi-procedural program $P$. This program has been taken from [50].

or in procedures that transitively call $P$ are marked, i.e. the traversal ascends from the procedure $P$ upwards the procedures that called $P$. In this phase, Parameter-out are not followed therefore we do not descend into the procedures that are called by $P$. Summary edges from Actual-in to Actual-out nodes cause nodes to be included in the slice that would be reached through the procedure call. Figure 2.24 computes the slice of the SDG shown in Figure 2.23 with respect to variable $z$ in the procedure Inc as a result of the first phase. We can see that all nodes that are part of the calling context that might affect the value of the variable $z$ are marked in this phase.

2. In the second phase, all edges except Parameter-in and call edges are followed backwards starting from all the marked nodes as a result of the first phase. Because Parameter-in and call edges are not followed, the traversal does not ascend into the calling procedure. Again, the summary edges simulate the effects the calling procedures. The marked nodes represent all nodes in the called procedures that induce summary edges. Figure 2.25 shows how the slice is computed with respect to the variable $z$ in the procedure Inc. All the nodes that are marked in the second phase are
Figure 2.23: System dependence graph (SDG) of the program in Figure 2.22. Boldface edges represent control dependences, arcs represent data-flow dependences, where dashed lines represent Parameter-in, Parameter-out and call edges.
added the ones marked as a result of the first phase. The nodes and edges introduced
as a result of the second phase are marked in boldface.

Figure 2.26 presents the final slice with respect to z in the procedure Inc.

2.6 Applications

2.6.1 Debugging

Program debugging was the main motivation behind the introduction of program slicing
by Weiser [85]. His motivation was a result of an observation he made when debugging
a program. He noted that programmers follow data and control dependencies to identify
and locate the program statements responsible for the error [89]. During debugging, a
programmer usually has a test case in mind which causes the program to fail. A program
slicer that is integrated into the debugger can be very useful in discovering the reason for
the error by visualising control and data dependences and by highlighting that statements
that are part of the slice. Variants of program slicing have been developed to further assist
the programmer: program dicing [63] identifies statements that are likely to contain bugs
by using information that some variables fail some tests while others pass all tests. Several
slices are combined with each other in different ways: the intersection of two slices contains
all statements that could lead to an error in both test cases; the intersection of a slice A
with a complement slice B excludes from slice A all statements that do not lead to an
error in the second test case. Another type of program slicing is program chopping [52].
It identifies statements that lie between two points n and m in the program and will be
affected by a change at n. This can be useful when a change at n causes an incorrect result
at m. Debugging should be focused on the statements between n and m that transmit the
change of n to m.
Figure 2.24: Slice of SDG in Figure 2.23 resulting after the first phase.
Figure 2.25: Slice of SDG in Figure 2.23 as resulting after the second phase.
Figure 2.26: Slice of SDG in Figure 2.23 as resulting after the first and the second phase.
A bug in a program can be detected as the result of the execution of the program with respect to some specific input. As result of this, program debugging was the main motivation of Korel and Laski [57] to think of another variant of program slicing, dynamic program slicing [57], which produces slices that preserve the behaviour of the original program with respect to a particular input. Dynamic slicing [57] produces more accurate and smaller slices in size than than those produced by using traditional static slicing [88]. For this reason, dynamic program slicing is a more suited technique to assist programmers in locating a bug, exhibited on a particular execution path of the program [1, 58].

### 2.6.2 Program Differencing

*Program differencing* [21], is a way of analysing an old and a new version of a program in order to determine which part of the new version represent syntactic and semantic changes. This information is useful as in incremental testing where only the components that have a different behaviour need to be tested. The main issue of program differencing consists of the partition of the components of the old and new version in a such a way that two components are in the same category only if they have equivalent behaviours. Program slicing can be used to identify *semantic* differences between two programs. This can be done in two stages:

1. We first find all the components of the two programs that have different behaviour. This can be done by comparing, using the dependence graphs, the backward slices of the old and new programs.

2. The second stage is the find a program that captures the revised semantics behaviour. This can be done by taking the backward slice with respect to a set of all of the affected program points determined in the first stage.
2.6.3 Program Integration

*Program integration* is a technique which consists of merging program variants [11, 6, 48]. The main motivation of the program integration technique is when different versions of a program have been made and a bug-fix, for example, is needed for all of them. Given a program *Base* and two different variants, A and B, obtained by modifying separate copies of the Base. Program integration will determine whether the modifications interfere or not. If an integrated program that incorporates both sets of changes as well as the portions of the *Base* which are preserved in both variants is not created [48], then *Program differencing* [21] is used to identify the changes in Variants A and B. The program integration algorithm discussed below compares slices in order to detect equivalent behaviours.

Horwitz et al. [48] used the static slicing algorithm for single-procedure programs as the basis for an algorithm that integrates changes in variants of a program. The algorithm consists of the following steps:

1. First, construct the program dependence graph (PDG), of the *Base*, A and B. Let $G_{Base}$, $G_A$ and $G_B$ represent the PDGs of the Base, A and B respectively.

2. The sets of affected points of $G_A$ and $G_B$ with respect to $G_{Base}$ are determined. These consist of vertices in $G_A(G_B)$ which have a different slice in $G_{Base}$.

3. A merged program dependence graph, $G_M$ is constructed from $G_A$, $G_B$ and the sets of affected points determined in (2).

4. Using $G_A$, $G_B$, $G_M$ and the sets of affected points computed in (2), the algorithm determines whether or not the behaviours of A and B are preserved in $G_M$. This is accomplished by comparing the slices with respect to the affected points of $G_A(G_B)$ in $G_M$ and $G_A(G_B)$. If different slices are found, the changes interfere and the integration cannot be performed.
5. If the changes in A and B do not interfere, then algorithm tests if $G_M$ is a feasible, i.e.,
   if it corresponds to some program. If this is the case, a program $M$ is constructed
   from $G_M$. Otherwise, the changes in A and B cannot be integrated.

### 2.6.4 Software Maintenance

The main challenges in software maintenance are to understand existing software, and
make new changes without introducing new bugs. Decomposition slicing [32] is a useful
tool in making a change to a piece of software without introducing unwanted side-effects.
It captures all computations of a variable’s value and is independent of a program location.
The decomposition slice for a variable $v$ is the union of slices taken at critical nodes with
respect to $v$. Critical nodes are the nodes that output the value of $v$ and the last node of
the program. The decomposition slices are computed for all variables of the program. The
decomposition slice for variable $v$ partitions the program into three parts:

**The independent part** contains all the statements of the decomposition slice, with respect
to $v$, that are not part of any decomposition slice taken with respect to another
variable.

**The dependent part** contains all statements of the decomposition slice, with respect to $v$,
that are part of another decomposition slice taken with respect to another variable.

**The complement parts** contains all statements that are not in the decomposition slice
taken with respect to $v$. The statements of the complement may nevertheless be part
of some other decomposition slice taken with respect to another variable. The comple-
ment must remain fixed after any change made to statements of the decomposition
slice.

In the same way the variable $v$ can be partitioned:
Changeable if all assignments to v are within the independent part.

Unchangeable if at least one assignment to v is in the dependent part. If the assignment has been modified, the new value will flow out of the decomposition.

Used if it is not used in the dependent or independent parts but in the complement. The maintainer may not declare new variables with the same name.

The modification can be made as follows:

- Statements in the independent part do not have any effect on the computation of the complement. Therefore, they can be deleted from a decomposition slice.
- Assignments to the changeable variables may be added anywhere in the decomposition slice.
- New control statements that surround any statements of the dependency part will cause the complement to change.

The maintainer who tries to change the code only has to consider the dependent and independent parts of the program. The complement part is guaranteed then to be unaffected by the modification and therefore there is no need to be retested [32]. The only parts to be retested after the modification are the dependent and the independent parts. There has been much work in using program slicing in software maintenance [13, 17, 31, 32].

2.6.5 Testing

After any modification, software has to be retested. A large number of test cases may be necessary, even after a small modification. Decomposition slicing can be used to reduce the program to a smaller one, which is easy to test, i.e. regression testing on the complement
is not needed. Only the dependent and independent parts need to be to be retested. Extensive work has been done to simplify testing, using program slicing [8, 35, 37, 45, 46]. In [46], we have illustrated the use of program slicing in partition testing. Using program slicing to determine components affected transitively by a change in a program point \( p \), Gupta et al. [35] introduced an algorithm for reducing the cost of regression testing. They use two variants of slices to achieve their goal. They first work out a backward slice starting from the program point \( p \) and record all the definitions of the variables used at \( p \). They then work out a forward slice starting from the same point. Any variable which is defined at \( P \) and used in this slice is recorded. Finally Def-Use pairs from a definition of the first slice to a use in the second might be affected by the change at \( p \) and hence, they must be retested.

### 2.7 Program Semantics and Program Slicing

Program semantics is the meaning of programming languages. Program semantics can be used to model program transformation techniques such as slicing in our case. Different kinds of program semantics have been derived for different applications [79, 15, 33]. Modelling different program manipulation techniques may require different kinds of program semantics.

#### 2.7.1 Standard Denotational Semantics

Denotational semantics [79], enables mathematical meaning to be given to programming languages. It combines mathematic rigour and notational elegance [76].
Definition 13 (Standard Denotational Semantics): In denotational semantics [79], a state, \( \sigma \in \Sigma \), is a mapping from program variables in \( I \) to values in a set \( V \).

\[
\Sigma_\bot = \Sigma \cup \bot = [I \mapsto V] \cup \bot.
\]

For example, the function \( \sigma = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\} \) is the state where the value of \( x \) is 1, the value of \( y \) is 2 and the value of \( z \) is 3. The meaning of a program is given by a function from states to states:

\[
\mathcal{M} : P \rightarrow \Sigma_\bot \rightarrow \Sigma_\bot.
\]

Where \( P \) is the set of all programs. \( \mathcal{M}[p]\sigma \) represents the final state after executing the program \( p \) in the initial state \( \sigma \). If the program \( p \) does not terminate in initial state \( \sigma \), then \( \mathcal{M}[p]\sigma \) has the special value \( \bot \), known as bottom. In standard semantics, in the bottom state all variables are deemed to have the value \( \bot \). The bottom state is the state that maps every variable name to \( \bot \). The final value of variable \( x \) after executing \( p \) in initial state \( \sigma \) is thus written \((\mathcal{M}[p]\sigma)x\).

Ordering on States

In the standard denotational semantics [79], the ordering on states is such that two distinct non-terminating states are incomparable and \( \bot \) is less than every state. The reason an ordering is required is that the meaning of loop is defined in terms of the least fixed point.

Evaluating Expressions

The meaning of an expression \( e \) is given by the function \( \mathcal{E} \). It evaluates an expression in state to give a value.
\[ \mathcal{E} : E \rightarrow \Sigma \mapsto V \]

where

\[ V = V \cup \{\perp\}. \]

**Strictness of \( \mathcal{E} \) in standard semantics**

A function is strict if it gives \( \perp \) when applied to \( \perp \). In standard semantics \( \mathcal{E} \) is strict. In other words, evaluating every expression in the \( \perp \) state will give the \( \perp \) value. Figure 2.27 shows the meaning of each construct of the language considered using the standard semantics.

### 2.7.2 Weakest Precondition and Strongest Postcondition Semantics

**Weakest Precondition Semantics**

The *weakest precondition* [24], \( wp \), is defined as follows: a program \( S \) and a postcondition \( P \), the *weakest precondition* [24] for \( S \) to establish \( P \), \( wp(S, P) \), is the weakest condition that must hold in the initial state to ensure that the program \( S \) will terminate in a state satisfying \( P \). A program which establishes \texttt{True} can terminate from any state. If \( wp(S, P) = \texttt{False} \), then there is no guarantee that \( S \) will terminate in a state satisfying \( P \), i.e. for any initial state \( S \) may do anything, including not terminate. A weakest precondition is the least restrictive precondition that will guarantee the postcondition. The *weakest precondition* and the *strongest postcondition* [62] proved to be useful in various areas of software development [62, 3, 65]. The *weakest precondition* semantics for a simple imperative language is now given in Figure 2.28.
\( \mathcal{M} : P \rightarrow \Sigma_\bot \rightarrow \Sigma_\bot \)

**skip statement**

\( \mathcal{M}[\text{skip}] = \lambda \sigma \cdot \sigma \)

**abort statement**

\( \mathcal{M}[\text{abort}] = \lambda \sigma \cdot \bot \)

Assignment statements

\( \mathcal{M}[x := e] = \lambda \sigma \cdot \sigma[x \leftarrow \mathcal{E}[e] \sigma] \)

(where the notation \( \sigma[x \leftarrow e] \) represents the function which is the same as \( \sigma \) except that \( x \) gets mapped to \( e \).)

Sequences of statements

\( \mathcal{M}[S_1; S_2] = \lambda \sigma \cdot \mathcal{M}[S_2](\mathcal{M}[S_1] \sigma) \)

**if statements**

\( \mathcal{M}[\text{if}(B) \ \text{then} \ S_1 \ \text{else} \ S_2] = \lambda \sigma \cdot \mathcal{E}[B] \sigma \rightarrow \mathcal{M}[S_1] \sigma, \mathcal{M}[S_2] \sigma \)

(where the notation \( e \rightarrow b, c \) is the expression which yields \( b \) if \( e \) is True and \( c \) if \( e \) is False).

**while loops**

\( \mathcal{M}[\text{while}(B) \ S] = \text{fix} (\lambda f \cdot \lambda \sigma \cdot \mathcal{E}[B] \sigma \rightarrow f(\mathcal{M}[S] \sigma, \sigma)) \)

Figure 2.27: Standard denotational semantics.
skip statement
\[ wp(\text{skip}, P) = P \]

abort statement
\[ wp(\text{abort}, P) = \text{False} \]

Assignment statements
\[ wp(x:=e, P) = P[x/e] \]
(\text{It is just } P \text{ where all the occurrences of } x \text{ are replaced by } e.)

Sequences of statements
\[ wp(S_1; S_2, P) = wp(S_1, wp(S_2, P)) \]
we first compute the weakest precondition of for \( S_2 \) to establish \( P \), i.e. \( wp(S_2, P) \). This in turn must be established by \( S_1 \), so the execution must start in a state where \( wp(S_1, wp(S_2, P)) \) holds.

if statements
\[ wp(\text{if}(B) \text{ then } S_1 \text{ else } S_2, P) = (B \implies wp(S_1, P)) \land (\neg B \implies wp(S_2, P)) \]
(if \( B \) is true then \( S_1 \) must establish \( P \) for which the precondition is \( wp(S_1, P) \).
if \( B \) is false then \( S_2 \) must establish \( P \).

while loops
\[ wp(\text{while}(B) S, P) = \exists n \in \mathbb{N} \cdot \psi_n(P) \]
where
\[ \psi_0(P) = \neg B \land P, \text{ and } \psi_{n+1}(P) = \psi_0(P) \lor wp(\text{if}(B) \text{ then } S \text{ else } \text{skip}, \psi_n(P)) \]
(if in the initial execution state \( \psi_0 = \neg B \land P = \text{True} \) then the \text{while loop will terminate without executing } S \text{ in a state where } P \text{ holds}.
\( \psi_n(P) \) is the weakest precondition that makes the \text{while loop statement establish } P \text{ in at most } n \text{ iterations.}

Figure 2.28: Weakest Precondition Semantics.
2.7 Program Semantics and Program Slicing

Strongest Postcondition Semantics

Given a precondition \( Q \) and a program \( S \), the strongest postcondition \([62]\), denoted \( sp(S, Q) \), is the strongest of the predicates on final program states that are reachable if the execution of \( S \) started in states that satisfy \( Q \) and it terminates. Given an initial state \( \sigma \) where \( Q \) holds, if the execution of \( S \) terminates on \( \sigma \) then the final state satisfies \( sp(S, Q) = \text{True} \). The strongest postcondition semantics for a simple imperative language is now given in Figure 2.29.

Specification-based Slicing

Lee et al. \([62]\) extend the definition of program slice to a specification-based slice using the concept of weakest precondition and strongest postcondition, called specification-based slicing. Specification-based slicing takes a specification, which is pre-postcondition pair \((P, Q)\). Let \( S_1 \) be a subprogram of \( S_2 \), we say \( S_1 \) is specification-based slice of \( S_2 \) with respect to the given pre-postcondition pair \((P, Q)\) if \( S_1 \) preserves the correctness of \( S_2 \) with respect to \((P, Q)\): If \( S_2 \) satisfies the given specification \((P, Q)\) then \( S_1 \) is.

**Definition 14** (Specification-based slice): Given two program \( S_1 \) and \( S_2 \), we say that \( S_1 \) is a specification-based slice of \( S_2 \) with respect to \((P, Q)\) if and only if

\[
(P \implies wp(S_2, Q)) \implies (P \implies wp(S_1, Q)) \text{ and } S_1 \text{ is a subprogram of } S_2.
\]

In \([62]\), Lee et al. also introduced precondition-based slicing and postcondition-based slicing.

**Definition 15** (Postcondition-based slice): Given two program \( S_1 \) and \( S_2 \), we say that \( S_1 \) is a postcondition-based slice of \( S_2 \) with respect to a postcondition \( Q \) if and only if

\[
(wp(S_1, Q) \equiv wp(S_2, Q)) \text{ and } S_1 \text{ is a sub-program of } S_2.
\]
skip statement  
\( sp(\text{skip}, Q) = Q \)

abort statement  
\( sp(\text{abort}, Q) = \text{False} \)

Assignment statements  
\( sp(x := e, Q) = Q[x/e] \)  
(It is just \( Q \) where all the occurrences of \( x \) are replaced by \( e \).)

Sequences of statements  
\( sp(S_1; S_2, Q) = sp(S_2, sp(S_1, Q)) \)

We first compute the strongest postcondition of for \( S_1 \) and \( Q \), i.e. \( sp(S_1, Q) \). This in turn must hold before executing \( S_2 \). We then compute the strongest postcondition for \( S_2 \) and \( sp(S_1, Q) \).

if statements  
\( sp(\text{if}(B) \text{ then } S_1 \text{ else } S_2, Q) = (sp(S_1, B \land Q)) \lor (sp(S_2, \neg B \land Q)) \)

while loops  
\( sp(\text{while}(B) \ S, Q) = \neg B \lor (\exists n \geq 0 sp((\text{if}(B) \ S)^n, Q)) \)

After execution of the loop, the loop guard is false(\( \neg B \)), and the second expression describes the effects of iterating the loop some number of times is true.

Figure 2.29: Strongest Postcondition Semantics.
Definition 16 (Precondition-based slice): Given two program $S_1$ and $S_2$, we say that $S_1$ is a precondition-based slice of $S_2$ with respect to a precondition $P$ if and only if

$$ (sp(S_1, P) \equiv sp(S_2, P)) \text{ and } (P \implies wp(S_2, \text{True})) \text{ and } S_1 \text{ is a subprogram of } S_2. $$

Program slicing algorithms are defined in terms of data-flow and control-flow dependencies. Program-dependence graphs [26] combine both data and control-dependencies. For this reason, program slicing techniques have been defined in terms of program dependence graphs [26]. There have been many efforts to give a formal semantics of these intermediate program representations [47, 75, 15, 47].

Horwitz et al. [47] show that program dependence graphs capture the program’s behaviour. They showed that if program dependence graphs of two programs, $P_1$ and $P_2$ are isomorphic then $P_1$ and $P_2$ are strongly equivalent, the standard meanings of $P_1$ and $P_2$ agree in all states $\sigma$ in $\Sigma$. Such equivalence makes it reasonable to try to develop a semantics for program dependence graphs that is consistent with the program semantics. Various attempts have been made to define a semantics of program dependence graph for different applications [75, 15].

Selke et al [75] presented a rewriting of the semantics of the program dependence graph of a simple procedural language. The rewriting captures the compiler writer’s and the programmer’s intuitive notion of how processing proceeds and thus provides processing information in addition to final results. This semantics represents computation steps as graph transformations. The dependence edges are used to ensure that statements are executed in the right order. The store, state, is embedded in the graph. When assignment statements are executed, the relevant portions of the graph are updated to reflect the new value of the corresponding variable. The evaluation of the predicate of an if statement
results in deletion of the part of the program representing the True or False branch, as appropriate. The evaluation of while statements results in the deletion of the body or creation of a copy of it as necessary.

For example, in program slicing, there is no guarantee that a slice will fail to halt when the original does. Traditional slicing algorithms sometimes introduce termination: the standard semantics of the program is thus, less defined than the semantics of some of its slices. Because of this it is unnatural to try to prove correctness of slicing properties using standard semantics [33]. This was observed and first discussed by Cartwright and Felleisen [15]. For example, the standard meaning of the variable \( x \) after executing the program \( P \) in Figure 2.30 is \( \bot \), where the variable \( x \) after executing the slice of \( P \) with respect to the variable \( x \), \( x:=1 \), has 1 as its value.

Furthermore, central to slicing is the concept of variable dependence-(neededness): the set of variables needed by a set of variables \( V \) in program \( P \). Intuitively, this is the set of variables whose initial value ‘may affect’ the final value of at least one variable \( v \) in \( V \) after executing \( P \). Our aim is to make the phrase ‘may affect’ semantically precise. In Section 3.2, Chapter 4 we will show how difficult it is, if it is not impossible, to define variable dependence in terms of standard semantics.

\[
x:=2;
while (x>0) y:=y+1;
x:=1;
\]

Figure 2.30: A non-terminating program \( P \).

Finally, for a semantics to be useful for modelling program slicing techniques, it has to be able to look beyond an infinite loop.
2.7.3 Lazy semantics of program dependence graphs—Cartwright and Felleisen

Cartwright and Felleisen [15] have defined a non-strict generalisation for the denotational semantics of program dependence graphs. They use a staging analysis to decompose the meaning function into two functions: a compiler function that transforms programs into code trees, which resemble program dependence graphs, and an interpreter function which provides an operational semantics for these code trees. The standard denotational semantics was made non-strict by making the functional update of stores non-strict. Cartwright and Felleisen used two new definitions of the update operator:

Lazy Update

\[
\sigma[v/x](y) = \begin{cases} 
\bot & \text{if } x = \bot \\
\sigma(y) & \text{if } y \neq x \land x \neq \bot \\
v & \text{if } y = x \land x \neq \bot 
\end{cases}
\]  

(2.1)

Lackadaisical Update

\[
\sigma[v/x](y) = \begin{cases} 
\sigma(y) & \text{if } y \neq x \\
v & \text{if } y = x \land \sigma(x) \neq \bot \\
\bot & \text{if } y = x \land \sigma(x) = \bot 
\end{cases}
\]  

(2.2)

The rest of this section will focus on the lazy update 2.1. \(M_{lazy}\) is the non-strict semantics derived from \(M\) by replacing the the standard update function by the lazy update 2.1. In lazy semantics, the value of an expression is never computed until the value is required from the store, i.e., until there is a data demand for the expression. Hence, an expression whose value is never computed. The propagation of data demands in lazy semantics is hidden by the use of a central store. The demands can be made by rewriting the semantic
function to eliminate the central store. The lazy semantics is defined as follow:

\[ M_{\text{lazy}} : \text{statement} \to \text{identifier} \to \text{store} \to \text{value}. \]

Instead of

\[ M_{\text{lazy}} : \text{statement} \to \text{store} \to (\text{identifier} \to \text{value}). \]

In Lazy semantics, \( M_{\text{lazy}} \), for each identifier \( x \) the final assignment to \( x \) is identified first. The semantics is then applied recursively to every identifier used in the final assignment to \( x \) to propagate the data dependence. Although this construction identifies data dependence between statements, control dependences must also be taken into account to determine the dependence relation between expressions and predicates in a program. To circumvent this problem Cartwright [15] introduced an additional control parameter into the semantic function definition. The control parameter, \( K \) represents the accumulated boolean value of all predicates that control the computation of the current statement. The resulting semantics is called the control and demand semantics.

A two stage analysis was then performed on the control and demand semantics. This, analysis separates the static portion of a semantic definition from the rest. To perform this two stage analysis, two functions are used: a static meaning function that performs the static computations and generates an intermediate representation graph representation for each identifier of the program, called code tree. Figure 2.31 illustrates the notation used in the static meaning function. Figure 2.32 shows the static meaning function of each construct of a simple procedural language, and an interpretation function that evaluates the intermediate code tree. Figure 2.33 shows the interpretation of code trees produced by the static meaning function. Data-nodes and control-nodes form the basic units of the intermediate graph representations: A data node is of the form \( \langle e, c_e, c_k \rangle \), where \( e \) is an expression in the program whose value is assigned to the variable of interest and \( c_e \) is a table that associates with each variable used in \( e \) the data-node required to compute its
value. $c_e$ implicitly captures the data dependencies for $e$ in the program. $c_k$ is a control node that dominates the evaluation of $e$. $c_k$ captures the control dependencies for $e$. A control node is of the form $(B, e, c_e, c_k)$, where $B$ is boolean tag that labels the control dependence from the boolean expression $e$. $c_e$ and corresponds to the data dependence of $e$ while $c_k$ corresponds to control dependence of $e$.

The static meaning function derived from the lazy demand and control semantics generates intermediate representations of the program for a given variable. The generated code contains all the information required to generate a program dependence tree which can be collapsed into a program dependence graph. The program dependence tree can be seen as a program dependence graph in which each path from the root to a leaf corresponds to the dependence relationships in a single possible execution of the program.

**Lazy semantics of Cartwright and Felleisen is not Substitutive**

**Definition 17: Substitutivity**

A semantic analysis is described as substitutive if, a sub-program $Q$ of a program $P$ can be replaced with another semantically equivalent sub-program, $Q'$, and guarantee that the resulting program $P'$ is semantically equivalent to our original program $P$.

Substitutivity of the semantics simplifies correctness proofs for the sorts of transformations described in this thesis and others, such as those used in amorphous slicing [38, 9, 42] where the program has to preserve only the semantics but not necessarily the syntax.

Although the lazy semantics of Cartwright and Felleisen [15] is able to look beyond an infinite loop, it loses precision for all variables defined in the body of an if or while statement in states where their corresponding predicate is evaluated to ⊥. This is due to the fact that the evaluation of any expression demands its controlled predicate to be evaluated first. As a result of this the lazy semantics of Cartwright and Felleisen [15] is not substitutive.
Notation

\[ A \otimes B = \{ \bot \} \cup \{ (a, b) \mid a \in A \setminus \{ \bot \} \cap a \in B \setminus \{ \bot \} \} \]

\[ A \oplus B = \{ \bot \} \cup \{ (T, a) \mid a \in A \setminus \{ \bot \} \} \cup \{ (T, b) \mid b \in B \setminus \{ \bot \} \} \]

\[ A_\bot = \{ \bot \} \cup \{ (T, a) \mid a \in A \} \]

\[ A^+ = A \otimes (A^+)_{\bot} \]

\[ < \ldots > : A \times B \rightarrow A \otimes B \]

\[ < a, b > = \begin{cases} 
(a, b) & \text{if } a \neq \bot \land b \neq \bot \\
\bot & \text{otherwise}
\end{cases} \]

\[ < e_1, e_2, \ldots, e_n > = < e_1, < e_2, \ldots > > \]

\[ [e_1, e_2, \ldots] = < e_1, (T, < e_2, (T, \ldots)) >> > \]

\[ \circ : A \times A^+ \rightarrow A^+ \]

\[ a_0 \circ [a_1, a_2, \ldots] = < a_0, < T, [a_1, a_2, \ldots] >> = [a_0, a_1, a_2, \ldots] \]

Figure 2.31: Notation used in static meaning function.
2.7 Program Semantics and Program Slicing

Domains

\[
\begin{align*}
\text{code-table} & = \text{identifier} \rightarrow \text{code} \\
\text{data-node} & = \text{expression} \odot \text{code-table} \odot \text{control-node} \\
\text{code} & = \text{data-node} \odot (\text{data-node} \odot \text{data-node}) \odot \text{data-node}^+ \\
C_k & \in \text{control-node} = \text{True} \odot \text{data-node} \odot \text{data-node}.
\end{align*}
\]

Static meaning function

\[\mathcal{G}_{a z y} : \text{statements} \rightarrow \text{identifier} \rightarrow \text{control-node} \rightarrow \text{code-table} \rightarrow \text{code}\]

Assignment statements

\[\mathcal{G}_{a z y}[x:=e] = \lambda ik\gamma. i = x \rightarrow e, \{(j, \gamma j) | j \in \text{Referenced}(e)\}, k \succ, \gamma i\]

Sequence of statements

\[\mathcal{G}_{a z y}[S_1; S_2] = \lambda ik\gamma. \mathcal{G}_{a z y}[S_2] ik(\lambda j. (\mathcal{G}_{a z y}[S_1]\ j k\gamma))\]

if statements

\[\mathcal{G}_{a z y}[\text{if}(B) \text{ then } S_1 \text{ else } S_2] = \]

\[\lambda ik\gamma. i \notin \text{Defined}(S_1) \cup \text{Defined}(S_2) \rightarrow \gamma i.\]

\[\begin{align*}
\text{let } K^+ &= < T, B, \{(j, \gamma j) | j \in \text{Referenced}(B)\}, K > \\
K^- &= < F, B, \{(j, \gamma j) | j \in \text{Referenced}(B)\}, K > \\
in &< (i \notin \text{Defined}(S_1) \rightarrow < \{(i, \lambda i)\}, K^+ >, \mathcal{G}_{a z y}[S_1] ik^+ \gamma), \\
&< (i \notin \text{Defined}(S_2) \rightarrow < \{(i, \lambda i)\}, K^- >, \mathcal{G}_{a z y}[S_2] ik^- \gamma) >.
\end{align*}\]

while statements

\[\mathcal{G}_{a z y}[\text{while } (B) \ S] = \]

\[\begin{align*}
\text{fix}(\lambda i. \lambda ik\gamma. i \notin \text{Defined}(S) \rightarrow \gamma i. \\
\text{let } K^+ &= < T, B, \{(j, \gamma j) | j \in \text{Referenced}(B)\}, K > \\
K^- &= < F, B, \{(j, \gamma j) | j \in \text{Referenced}(B)\}, K > \\
in &< i, \{(i, \lambda i)\}, K^- > \circ (fk^+(\lambda j. \mathcal{G}_{a z y}[S]\ j k^+ \gamma))).
\end{align*}\]

Figure 2.32: Static meaning function.
\[ \mathcal{I} : \text{code} \rightarrow \text{store}^\top \rightarrow \text{value}^\top \]

\[ \mathcal{I}[[c_1, c_2, \ldots]] = \lambda \sigma. \mathcal{I}[c_1] \sigma \sqcup \mathcal{I}[[c_2, \ldots]] \sigma \]

\[ \mathcal{I}[< c_1, c_2 >] = \lambda \sigma. \mathcal{I}[c_1] \sigma \sqcup \mathcal{I}[c_2] \sigma \]

\[ \mathcal{I}[< e, c_1, c_k >] = \lambda \sigma. \mathcal{I}_c[c_k] \sigma \rightarrow \mathcal{E}[e] \{ (j, \mathcal{I}[c_j] \sigma) \mid (j, c_j) \in c_e \}, \bot \]

where, \( \mathcal{I}_c : \text{control-node} \rightarrow \text{store}^\top \rightarrow \text{bool}^\top \)

\[ \mathcal{I}_c[\text{true}] = \lambda \sigma. T \]

\[ \mathcal{I}_c[< T, e, c_1, c_k >] = \lambda \sigma. \mathcal{I}_c[c_k] \sigma \cap \mathcal{E}[e] \{ (j, \mathcal{I}[c_j] \sigma) \mid (j, c_j) \in c_e \} \]

\[ \mathcal{I}_c[< F, e, c_1, c_k >] = \lambda \sigma. \mathcal{I}_c[c_k] \sigma \cap \neg \mathcal{E}[e] \{ (j, \mathcal{I}[c_j] \sigma) \mid (j, c_j) \in c_e \} \]

Figure 2.33: Interpretation function \( \mathcal{I} \).
Figure 2.34: $P_1$ and $P_2$ do not have to same lazy semantics w.r.t. $x$.

For example, in the program $P_1$ defined in Figure 2.34, the value of the variable $y$ “after” executing the infinite loop is undefined, and thus, so is the value of the if predicate. Therefore, the final value of the variable $x$ demands the evaluation of an undefined predicate and hence, using the lazy semantics by Cartwright and Felleisen [15], the final value of the variable $x$ after executing the program $P_1$ is $\bot$. The assignment, $x := x$ and $\text{skip}$ have the same lazy semantics, then if this semantics is substitutive, then, the program $P_1$ and $P_2$ should be equivalent with respect to it. This however, is not the case as the final value when executing the program $P_2$ is 1, which is different from $\bot$ in the case of $P_1$.

2.7.4 Transfinite semantics—Giacobazzi and Mastroeni

Giacobazzi and Mastroeni [33] argued that if a semantic analysis is to be useful for modelling types of program manipulation such as slicing it should be able to capture semantic information ‘beyond infinite loops’ and be compositional, not in the standard sense but only, with regard to sequence of statements, i.e. it should sequential. They use transfinite
state traces of programs \cite{55} and showed the existence of such semantics using domain equations. They introduced a non-standard semantics, called \textit{transfinite semantics}, using a metric structure on their value domains. Their semantics is not given explicitly for each construct of the language. The \textit{transfinite semantics} of a program is defined in terms of the set of all \textit{possibly transfinite computations}: whose length can be any ordinal, finite or infinite.

\begin{verbatim}
x:=1;
y:=1;
while (y>0) y:=y+1;
\end{verbatim}

Figure 2.35: Program in which the transfinite meaning of $y$ is $\omega$

For example, the semantic traces in transfinite semantics of the program in Figure 2.35 is given as follows:

\[ ((x, 1), (y, 1) \rightarrow ((x, 1), (y, 2)) \rightarrow ((x, 1), (y, 3)) \rightarrow \cdots \rightarrow ((x, 1), (y, n)) \rightarrow \cdots ((x, 1), (y, \omega)) \]

Where $\omega$ is the first infinite ordinal. After executing the first infinite loop, the variable $y$ has the value $\omega$.

\textbf{The Transfinite Semantics is not Substitutive}

Giacobazzi and Mastroeni \cite{33} have illustrated the importance of sequentiality\footnote{Compositional with regards to sequences only.} of semantics. Nothing is said about substitutivity. Unsurprisingly their transfinite semantics is not substitutive. If an assignment to a variable $x$ is controlled by an undefined predicate\footnote{The predicate is evaluated to $\bot$.}, then the transfinite semantics will map $x$ to $\bot$. This implies that the transfinite semantics of
Figure 2.36: $P_1$ and $P_2$ do not have the same transfinite semantics w.r.t. $x$.

Giacobazzi and Mastroeni [33] is not substitutive$^3$: we cannot replace a part of program with an equivalent program and preserve the semantics of the original program. Unlike the lazy semantics of Cartwright and Felleisen [15], the two programs $P_1$ and $P_2$ in Figure 2.34 have the same transfinite semantics as the value of the variable $y$ after executing the first infinite loop is $\omega$ which is always greater than 0. Now consider the two programs $P_1$ and $P_2$ in Figure 2.36. $P_1$ and $P_2$ do not have the same transfinite semantics as the assignment $x:=x$ is controlled by an undefined predicate.

One of the main contributions of this thesis is to give a denotational definition of a non-strict semantics, which is substitutive, preserved by slicing algorithms and which is consistent with the standard semantics for terminating programs.

$^3$Private communication
2.7.5 Operational Semantics

Like denotational semantics, operational semantics is a way to give meaning to computer programs in a mathematically rigorous way. The operational semantics for a programming language describes how a valid program is interpreted as sequences of computational steps. A common way to rigorously define the operational semantics is to provide a state transition system for the language of interest. Such a definition allows a formal analysis of a language, permitting the study of relations between programs. Defining operational semantics through a state transition system is usually done by giving an inductive definition of the set of possible transitions. This usually takes the form of a set of inference rules which define the valid transitions in the system. Intuitively, operational semantics can be thought of as formally defining a compiler from the language being defined into a lower level well defined language. Similarly, denotational semantics is more closely related to an interpreter as it is defined in terms of states.

Structural Operational Semantics

In Structural operational semantics [69] the emphasis is on the individual steps of the execution. The transition relation is of the form

\[ <S, \sigma> \implies \gamma \]

where \( \gamma \) is of the form \( <S', \sigma'> \implies \gamma \) or of the form \( \sigma' \). The transition is the first step of the execution of \( S \) from the state \( \sigma \). \( ^1 \)

If \( \gamma \) is of the form \( <S', \sigma'> \) then the execution of \( S \) on state \( \sigma \) is not completed and the remaining computation is expressed by the intermediate configuration \( <S', \sigma'> \).

If \( \gamma \) is of the form \( \sigma' \) then the execution of \( S \) from the state \( \sigma \) is completed and the final state is \( \sigma' \).
If there is no $\gamma$ such that $<S, \sigma> \implies \gamma$ then $<S, \sigma>$ is stuck.

Operational semantics of a while language is given in Figure 2.37.

The rules $[\text{comp}^1_{sos}]$ and $[\text{comp}^2_{sos}]$ shows that the execution $S_1; S_2$ in a state $\sigma$ involves executing $S_1$ in $\sigma$ first. There are two possibilities:

1. If the execution of $S_1$ in $\sigma$ has not been completed. In this case the execution of $S_1$ has to be completed before the execution of $S_2$ can start. This shown by the rule $[\text{comp}^1_{sos}]$ in Figure 2.37. If the result of executing the first step $<S_1, \sigma>$ is equivalent the an intermediate configuration $<S', \sigma'>$ then the next configuration is $<S'; S_2, \sigma'>$.

2. If the execution of $S_1$ in $\sigma$ has been completed then the execution of $S_2$ can start. This case is shown by $[\text{comp}^2_{sos}]$ in Figure 2.37. If the result of executing $S_1$ in $\sigma$ is $\sigma'$ the execution of $S_1; S_2$ in $\sigma$ is just the execution of $S_2$ in $\sigma'$.

There are two rules for if statements: $[if^T_{sos}]$ if the then branch is selected and $[if^E_{sos}]$ is the else branch is selected instead.

The rule for the execution of while loops

$$[\text{while}_{sos}] \quad <\text{while}(B) \; \text{do} \; S, \sigma> = <\text{if}(B) \; \text{then} \; S; \text{while}(B) \; \text{do} \; S, \; \sigma>$$

shows that the first step to unfold the while loop one level and rewrite the rule as a conditional.

**Operational Semantics and Slicing**

Operational semantics has been used to define slicers of Hardware description languages. In [51], an approach to slicing the hardware language VHDL is presented based on the
skip statement

\[\text{[skip} \_ s_{sos}] \quad \langle \text{skip}, \sigma \rangle \implies \sigma\]

Assignment statements

\[\text{[ass} \_ s_{sos}] \quad \langle x := e, \sigma \rangle \implies \sigma[x \leftarrow \mathcal{E}[e]\sigma]\]

Sequences of statements

\[\text{[comp} \_ 1^{l_{sos}] \quad \langle S_1, \sigma \rangle \implies \langle S_1', \sigma' \rangle \implies \langle S_1; S_2, \sigma' \rangle\]

\[\text{[comp} \_ 2^{l_{sos}] \quad \langle S_1, \sigma \rangle \implies \sigma' \implies \langle S_1; S_2, \sigma' \rangle\]

if statements

\[\text{[if} \_ s_{sos}] \quad \langle \text{if}(B) \text{ then } S_1 \text{ else } S_2, \sigma \rangle \implies \langle S_1, \sigma \rangle \text{ if } \mathcal{E}[B] = \text{True}\]

\[\text{[if} \_ s_{sos}] \quad \langle \text{if}(B) \text{ then } S_1 \text{ else } S_2, \sigma \rangle \implies \langle S_2, \sigma \rangle \text{ if } \mathcal{E}[B] = \text{False}\]

while loops

\[\text{[while} \_ s_{sos}] \quad \langle \text{while}(B) \text{ do } S, \sigma \rangle \implies \langle \text{if}(B) \text{ then } S; \text{while}(B) \text{ do } S \text{ else } \text{skip}, \sigma \rangle\]

Figure 2.37: Structural operational semantics for while language.
operational semantics of VHDL.

In [71], a form of dynamic slicing of declarative languages is defined in terms of the operational semantics of these languages. They use an extended form of the standard operational semantics to capture control informations.

In [84], Martin Ward uses the operational semantics of his Wide Spectrum Language, WSL to define an *operational slice*. An operational slice, as the name suggests, is one which preserves operational semantics.
Chapter 3

Neededness—Variable Dependence

Variable dependence (neededness) is essential for program slicing [43, 86]. Weiser’s slicing algorithm [86, 87] consists of computing data flow information over the control flow graph of a program. Hausler [43] redefines Weiser’s algorithm for a simple language of assignments, conditionals and while loops. However, unlike Weiser, Hausler does not use any intermediate graph representation such as control flow graphs [88] or program dependence graphs [73]. He presents a denotational slicing algorithm directly in terms of the syntax of the language. His algorithm illustrates the importance of variable dependence in slicing.

3.1 Hausler’s Slicing Algorithm

Hausler [43] uses two auxiliary functions to define his denotational slicing algorithm:

\[ N_h(P, V), \text{ the Hausler needed set with respect to the set of variables, } V, \text{ of program } P. \]

\[ S_h(P, V), \text{ the Hausler slice of program } P \text{ with respect to the set of variables } V. \]

3.1.1 Hausler Needed—\( N_h \)

\[ N_h : S \times \mathcal{P}(I) \rightarrow \mathcal{P}(I) \]
3.1 Hausler’s Slicing Algorithm

\[
\begin{align*}
x &:= y; \\
x &:= x + 1;
\end{align*}
\]

Figure 3.1: Needed variables (data dependence)

where,

\[
\begin{align*}
I &= \text{the set program variables and,} \\
S &= \text{the set of programs.}
\end{align*}
\]

Hausler’s algorithm produces the set \( N_h(P, V) \) of variables needed by elements of \( V \) when executing the program \( P \). \( N_h(P, V) \) is the set of variables for which the initial values *may affect* the final values of some elements in \( V \) when executing the program \( P \). The reason that the phrase ‘*may affect*’ is used rather than ‘*do affect*’ is that the set of *semantically needed* variables (defined in section 3.3) is not computable. Hausler’s Algorithm, in order for it to produce correct slices, is conservative in the sense that \( N_h(P, V) \) contains all semantically needed variables but also possibly some more variables which are *not really* semantically needed. This is because Hausler’s algorithm uses *Def-Ref* abstraction to compute the needed set of variables, \( N_h(P, V) \) \[43\].

Before giving a formal semantic definition of neededness, we present some examples of Hausler-Neededness. In the program in Figure 3.1, the variable \( x \) needs \( y \) since the final value of \( x \) depends upon the initial value of \( y \). This corresponds to data dependence of \( x \) on \( y \). So

\[
N_h(P, \{x\}) = \{y\}.
\]

We now consider the program in Figure 3.2. Here the variable \( x \) needs \( y \) since the final value of \( x \) depends upon the initial value of \( y \) and \( x \). This corresponds to control dependence of
3.1 Hausler’s Slicing Algorithm

if (x>y)
   then x:=2;
else x:=3;

Figure 3.2: Needed variables (control dependence)

y:=z+k;
if (x>y)
   then x:=2;
else x:=3;

Figure 3.3: Needed variables (data and control dependence)

x on both x and y. In this case;

\[ N_h(P, \{x\}) = \{x, y\}. \]

Hausler-neededness can arise out of a combination of data and control dependencies as in Figure 3.3. Here:

\[ N_h(P, \{x\}) = \{x, z, k\}. \]

Figure 3.4: Hausler needed is not the same as Semantically Needed.
In Figure, 3.4, we see an example where \( N_h(P, \{ x \}) = \{ x, y \} \) but the final value of \( x \) is always 3 irrespective of the initial state. This is an example of the over conservativeness produced by Hausler’s algorithm and other algorithms that work at the level of data and control flow analysis.

We now give Hausler’s algorithm for computing \( N_h(P, \{ x \}) \) for each construct of a simple procedural programming language.

### 3.1.2 The Algorithm for \( N_h \)

1. **skip statements**

   \( N_h(\text{skip}, V) = V. \)

2. **abort statements**

   \( N_h(\text{abort}, V) = V. \)

3. **Assignment statements**

   \[
   N_h(x:=e, V) = \begin{cases} 
   V & \text{if } x \notin V \\
   (V \setminus \{x\}) \cup \text{Referenced}(e) & \text{otherwise.}
   \end{cases}
   \]

   Where \( \text{Referenced}(e) \) is the set of variables referenced by \( e \).

   If \( x \) is not an element of \( V \), the assignment \( x:=e \) does not influence any variable in \( V \) and \( V \) ‘flows through’ unchanged. If \( x \) is in \( V \), we transform the set flowing through \( x:=e \) by removing \( x \) and adding all the variables referenced by the expression \( e \).

4. **Sequence of statements**

   \( N_h(S_1; S_2, V) = N_h(S_1, N_h(S_2, V)). \)

   To work out the needed set of variables of a sequence \( S_1; S_2 \) with respect to a set of variables \( V \), first the set of needed variables \( W \) of \( S_2 \) with respect to \( V \) is computed
and following this, the needed set of $S_1$ with respect to $W$ is computed. In this way, the set of needed variables ‘flows backwards’ through the program.

5. **if** statements

$$N_h(\text{if } (B) S_1 \text{ else } S_2, V) = \begin{cases} V & \text{if } S_h(S_1, V)_{i=1,2} = \text{skip} \\ \bigcup_{i=1}^2 (N_h(S_i, V)) \cup \text{Referenced}(B) & \text{otherwise.} \end{cases}$$

If neither the True or False branches of an if statement, affect the elements in $V$ then, the set of needed variables of this statement is just $V$. Otherwise, the needed set of variables is the union of the needed sets of its components together with the variables referenced by the predicate. The referenced variables correspond to control dependence [44].

6. **while** loops

$$N_h(\text{while } (B) S, V) = \bigcup_{n \geq 0} \delta^n(\text{if } (B) S \text{ else skip}, V).$$

Where $\delta^0(P, V) = V$ and $\delta^{n+1}(P, V) = N_h(P, \delta^n(P, V))$.

For loops of the form $[\text{while}(B) S]$, the needed set of variables is produced by repeatedly calculating the needed set of variables of $[\text{if}(B) \text{ then } S \text{ else } \text{skip}]$ and feeding this value backwards until there is no further change.

### 3.1.3 Hausler’s Slicing Algorithm—$S_h$

$$S_h : S \times \mathcal{P}(I) \rightarrow S.$$  

Hausler uses the function $N_h$, defined above, to give a denotational algorithm for slicing. The Hausler slice of a program $P$ with respect to a set of variables of interest $V$, $S_h(P, V)$, is now given for each construct of simple procedural programming language.
1. **skip statements**

\[ S_h(\text{skip}, V) = \text{skip}. \]

2. **abort statements**

\[ S_h(\text{abort}, V) = \text{skip}. \]

3. **Assignment statements**

\[ S_h(x:=e, V) = \begin{cases} 
\text{skip} & \text{if } x \notin V \\
 x := e & \text{otherwise.} 
\end{cases} \]

If \( x \) is in not \( V \), the assignment \( x:=e \) does not affect any variable in \( V \) and the assignment is removed from the slice. However, when \( x \) is in \( V \) the assignment \( x:=e \) is kept in the slice as the value of the variable \( x \) may be affected.

4. **Sequence of statements**

\[ S_h(S_1; S_2, V) = S_h(S_1, N_h(S_2, V)); S_h(S_2, V). \]

When slicing a sequence \( S_1; S_2 \) with respect to a set of variables \( V \), first the needed set \( W \), of \( S_2 \) with respect to \( V \) is computed and following this, the needed set of \( S_1 \) with respect to \( W \) is computed. In this way, the needed set ‘flows backwards’ through the program being transformed ‘as it goes’. As the needed set ‘flows through’ each statement, the statement is sliced.

5. **if statements**

\[ S_h(\text{if } (B) \ S_1 \ \text{else } S_2, V) = \begin{cases} 
\text{skip} & \text{if } S_h(S_i, V)_{(i=1,2)} = \text{skip} \\
 \text{if } (B) S_h(S_1, V) \ \text{else } S_h(S_2, V) & \text{otherwise.} 
\end{cases} \]

The true and false parts are both sliced. The conditional is kept if and only if either of these slices is non-empty.

6. **while loops**
\[ S_{h}(\text{while}(B) \ S, V) = \begin{cases} \text{skip} & \text{if } S_{h}(S, V) = \text{skip} \\ \text{while}(B) \ S_{h}(S, N_{h}(\text{while}(B) \ S, V)) & \text{otherwise}. \end{cases} \]

If the body of the while loop does not affect any variable in \( V \) the whole while loop is removed. Otherwise the loop is left but its body becomes the slice of \( S \) with respect to \( N_{h}(\text{while}(B) \ S, V) \).

## 3.2 Neededness

One of the aims of this thesis is to prove correctness of Hausler’s algorithm with respect to Weiser’s semantic definition of a slice \([86, 87]\); i.e. when a program terminates, its corresponding slice produced by Hausler’s algorithm must preserve the standard semantics of the original program with respect to the slicing criterion.

In the previous section we discussed how, due to the non-computability of semantic neededness, Hausler’s algorithm produced a conservative approximation to the set of semantically needed variables. By a conservative approximation we mean that all semantically needed variables are included by the algorithm but possibly more.

One of our aims is to try to make the notion of semantic neededness completely precise. Clearly, in order to do this we will have to appeal to program semantics. For a semantic definition of neededness to be useful it has to satisfy the following intuitive properties:

1. It must be partially discriminating.
2. It must be Hausler-consistent.
3. It must be sub-sequential.

We now discuss each of these properties in turn.
3.2.1 Partially Discriminating

**Definition 18 (PSSD):**

We define the pair \((x, y)\), to be in the Partial Standard Semantic Discrimination relation with respect to \(P\), \(PSSD(P)\), if and only if the final value of \(x\) depends on the initial value of \(y\) when executing program \(P\). That is, there exists two initial states, differing only on \(y\), such that executing \(P\) terminates from these initial states and gives rise to different final values of \(x\).

**Definition 19 (Partially Discriminating):**

A relation \(R\) is partially discriminating if it contains the Partial Standard Semantic Discrimination Relation.

Saying that neededness, \(N\), should be partially discriminating means that if there exists two initial states, differing only on \(y\), such that executing \(P\) terminates on these initial states and gives rise to different final values of \(x\) then \(y \in N(P, \{x\})\). Later on in Section 3.3.2, we will also define totally discriminating in terms of a Total Standard Semantic Discrimination (TSSD) relation where non-termination is treated as a special value.

3.2.2 Hausler—consistent

As we explained earlier, Hausler in his algorithm uses Def-Ref abstraction to compute the needed set of variables. This is conservative approximation and therefore, it will always be examples where Hausler-needed variables are not semantically needed.

The semantic definition of neededness, if it is to be useful for our purposes, will have to be contained in Hausler-needed. That is, all variables that are semantically needed must also be Hausler-needed.
3.2 Neededness

![Programs P, Q, and P;Q](image)

\[ \mathbb{N}(P, \{x, y\}) = \{y\} \]
\[ \mathbb{N}(Q, \{x\}) = \{x, y\} \]
\[ \mathbb{N}(P; Q, \{x\}) = \emptyset. \]

Figure 3.5: \( \mathbb{N}(P; Q, \{x\}) \subset \mathbb{N}(P; \mathbb{N}(Q, \{x\})). \)

So, one of our first aims is to find a semantic definition, \( \mathbb{N} \), of neededness satisfying:

\[ \mathbb{N}(P, V) \subseteq \mathbb{N}_h(P, V). \]

3.2.3 Sub-sequentiality of Neededness

We define a function of \( f \) of type \( S \times T \rightarrow T \) to be *sequential* if and only if:

\[ f(P; Q, t) = f(P, f(Q, t)). \]

By definition, Hausler neededness, given in Section 3.1.1, is sequential. i.e.

\[ \mathbb{N}_h(P; Q, V) = \mathbb{N}_h(P, \mathbb{N}_h(Q, V)). \]

It would be wrong to expect our semantic neededness to be sequential. This can be seen from Figure, 3.5 where \( \mathbb{N}(P; Q, \{x\}) = \emptyset \), and \( \mathbb{N}(P; \mathbb{N}(Q, \{x\})) = \{y\} \).

Although we cannot expect our definition of semantic-neededness to be sequential we do require it to have a weaker property that we call *sub-sequentiality*. A function \( f \) of type
$S \times T \rightarrow T$ is to be sub-sequential if and only if

$$f(P; Q, t) \subseteq f(P, f(Q, t)).$$

We require:

$$N(P; Q, V) \subseteq N(P, N(Q, V)).$$

Without sub-sequentiality the unhappy situation could arise where variable $x$ is needed by $v$ in $P; Q$, but for all variables $z$ needed by $v$ in $Q$, $x$ is not needed by $z$ in $P$. This contradicts any reasonable intuitive understanding of semantic neededness. It turns out that the sub-sequentiality of neededness is used frequently in many of the proofs in this thesis.

In the following section, we attempt to define neededness in terms of standard semantics. We demonstrate the difficulties in using the standard semantics to define neededness which satisfies the above three properties. It is these difficulties that lead us later on, in Chapter 4, to introduce a new lazy semantics, which unlike the semantics of Cartwright and Felleisen [15] is denotational and substitutive.

### 3.3 Neededness and Standard Semantics

In the previous section we have shown that any semantic definition of neededness has to be partially discriminating, Hausler-consistent and sub-sequential. In this section, we will show that the standard semantics is not sufficient to give a semantic definition of neededness that satisfies these three properties.

A discussion of the standard denotational semantics is given in the literature survey in Chapter 2. Figure 2.27 on page 68 shows the standard denotational semantics for each construct of the language considered.
Using Kleene’s Theorem [64] the least fixed point of a monotonic functional is the limit of an ascending chain of functions. From this it follows that:

**Lemma 3.3.1:** \[ \mathcal{M} \left[ \text{while } (B) \ S \right] = \bigcup_{i=0}^{\infty} F_i \]

Where: \[ F_0 = \lambda \sigma. \bot, \quad F_{i+1} = \lambda \sigma. \mathcal{E} \left[ B \right]_{\sigma} \rightarrow F_i(\mathcal{M} \left[ S \right]_{\sigma}), \ \sigma. \]

and, \[ F_i \subseteq F_{i+1} \ \forall i \geq 0. \]

### 3.3.1 Unfolding of while Loops

In [20], unfoldings of a while loop were defined. The \( n^{th} \) unfolding of the loop, \( \mathcal{W}_n \), ‘agreed’ with the loop in all states where the loop terminates in \( n \) or fewer iterations. The meaning of the \( n^{th} \) unfolding when applied to any other state is \( \bot \). The \((n+1)^{th}\) unfolding is defined recursively in terms of the \( n^{th} \) unfolding below:

**Definition 20 (Unfoldings):**

\[ \mathcal{W}_0(B, S) = \text{abort} \]
\[ \mathcal{W}_{n+1}(B, S) = \text{if } (B) \text{ then } S; \mathcal{W}_n(B, S) \text{ else skip} \]

for example the unfoldings of while(\( x > 0 \) \) \( x:=x+y; \) statement are defined as follows:

\[ \mathcal{W}_0 = \text{abort}, \]
\[ \mathcal{W}_1 = \text{if } (x > 0) \text{ then } x:=x+y; \text{ abort else skip} \]
\[ \mathcal{W}_2 = \text{if } (x > 0) \text{ then } x:=x+y; \mathcal{W}_1 \text{ else skip} \]
\[ \vdots \]
\[ \mathcal{W}_n = \text{if } (x > 0) \text{ then } x:=x+y; \mathcal{W}_{n-1} \text{ else skip} \]

We now show that the meaning of the while loop is the limit of the meanings of its unfoldings.

**Lemma 3.3.2:** \[ \mathcal{M} \left[ \text{while } (B) \ S \right] = \bigcup_{i=0}^{\infty} \mathcal{M} \left[ \mathcal{W}_i(B, S) \right] \]
Proof. From Lemma 3.3.1, we only need to prove that the lazy meaning of the \( i \)th unfolding is the same as \( F_i \). We prove this lemma by induction on \( i \). The base case is trivial as \( \mathcal{M}[\mathcal{W}_0(B, S)]_\sigma = \mathcal{M}[\text{abort}]_\sigma = \bot_\sigma = F_0(\sigma) \) for all \( \sigma \in \Sigma \). Induction hypothesis: Assume for all \( \sigma \in \Sigma \), \( \mathcal{M}[\mathcal{W}_i(B, S)]_\sigma = F_i(\sigma) \) and show that for all \( \sigma \in \Sigma \), \( \mathcal{M}[\mathcal{W}_{i+1}(B, S)]_\sigma = F_{i+1}(\sigma) \).

\[
\begin{align*}
\mathcal{M}[\mathcal{W}_{i+1}(B, S)]_\sigma &= \mathcal{M}[\text{if } (B) \ S; \mathcal{W}_i(B, S) \text{ else } \text{skip}]_\sigma \quad \text{(By definition)} \\
&= \mathcal{E}[B]_\sigma \rightarrow \mathcal{M}[[S]; \mathcal{W}_i(B, S)]_\sigma, \sigma \quad \text{(By if statement rule)} \\
&= \mathcal{E}[B]_\sigma \rightarrow \mathcal{M}[\mathcal{W}_i(B, S)](\mathcal{M}[S])_\sigma, \sigma \quad \text{(By sequence rule)} \\
&= \mathcal{E}[B]_\sigma \rightarrow F_i(\mathcal{M}[S])_\sigma, \sigma \quad \text{(By Induction hypothesis)} \\
&= F_{i+1}(\sigma).
\end{align*}
\]

Hence, for all \( i \geq 0 \), \( \mathcal{M}[\mathcal{W}_i(B, S)] = F_i \). From which it follows immediately that the meaning of a while loop is the limit of the meanings of its unfoldings.

One of the aims of this thesis is to prove correctness of Hausler’s slicing algorithm [43]. To achieve our goal, we need to define neededness semantically. Elaborating on the desiderata of Section 3.2, we are looking for a semantic definition of neededness which must satisfy:

1. \( \mathbb{N}(P, V) \) should be partially discriminating, i.e. it should contain the \( \text{PSSD} \) relation (see Definition 22);

2. is Hausler-consistent i.e. \( \mathbb{N}(P, V) \subseteq \mathbb{N}_h(P, V) \); and

3. is \textit{sub-sequential}, i.e. \( \mathbb{N}_h(P; Q, V) \subseteq \mathbb{N}_h(P, \mathbb{N}_h(Q, V)) \).

Below we demonstrate the difficulties in attempting to define neededness using standard program semantics.
3.3.2 Semantic Discrimination

Having given the standard denotational semantics, it is very easy to define $PSSD$ and Total Standard Semantic Discrimination ($TSSD$) formally.

**Definition 21:** $(x, y) \in TSSD(P)$ if and only if there exists two states $\sigma_1$ and $\sigma_2$ which differ only on $y$ such that:

$$\mathcal{M}[P]_{\sigma_1} x \neq \mathcal{M}[P]_{\sigma_2} x.$$

**Definition 22:** $(x, y) \in PSSD(P)$ if and only if there exists two states $\sigma_1$ and $\sigma_2$ which differ only on $y$ such that:

$$\bot \neq \mathcal{M}[P]_{\sigma_1} x \neq \mathcal{M}[P]_{\sigma_2} x \neq \bot.$$

**Lemma 3.3.3:** $PSSD(P) \subseteq TSSD(P)$ for any program $P$.

*Proof.* Let $x, y$ be two variables with $(x, y)$ in $PSSD(P)$. We show that $(x, y)$ is in $TSSD(P)$. By Definition 22, it follows that there exists two states $\sigma_1$ and $\sigma_2$ which differ only on $y$ such that:

$$\bot \neq \mathcal{M}[P]_{\sigma_1} x \neq \mathcal{M}[P]_{\sigma_2} x \neq \bot.$$

Thus, the result follows by a straightforward application of Definition 21. \qed

A semantic definition of neededness in terms of the standard semantics is difficult if it is not impossible. Giacobazzi and Mastroeni [33] claim that any semantics that cannot look beyond an infinite loop is not useful for proving the correctness of program slicing algorithms. This is probably due to the fact that Hausler’s and Weiser’s Slicing Algorithm [85, 88] and other slicing algorithms [49, 21] introduce termination. And thus, slicing algorithms do not preserve the projection of the standard semantics on the variables of interest. For example the Program $P_{3,6}$ in Figure 3.6 is a non-terminating program. Where the program $P'_{3,6}$ is its corresponding slice with respect to the variable $x$. Clearly, the original program and its
1: x:=1;
2: while (x>0) {y:=y+1;}
3: x:=2;

Program $P_{3,6}$.

1: 
2: 
3: x:=2;

$P'_{3,6}$: Slice of $P_{3,6}$ w.r.t. the variable x.

Figure 3.6: A non-terminating program $P_{3,6}$ and its slice w.r.t. the variable x.

slice do not have the same standard semantics.

All these difficulties, are caused by non termination. As a result of this the need for some kind of non-strict semantics became our prime priority. In the following chapter, a new lazy semantics is given. This semantics is different from all others in the literature. It is defined denotationally for each construct of the language. As it will be shown, unlike the standard semantics, it enables us to give new semantic definition of neededness in a very natural manner.
Chapter 4

Lazy Semantics for Program Slicing

The fact that a slice does not preserve a projection of the standard semantics means it is very difficult to attempt to develop a theory of slicing that enables us to prove correctness of slicing algorithms directly in terms of standard semantics [33].

In this chapter a new lazy semantics is defined. This lazy semantics is defined denotationally, just as with standard semantics, in terms the syntax of the language. We prove that our lazy semantics agrees with the standard semantics for terminating programs. In addition to this a new semantic definition of a slice is given in terms of this semantics. This definition is consistent with Weiser’s definition but stronger than it where a program and its slice only need to agree in states where the original terminates: if $T$ is a slice of $P$ with respect to our new definition of a slice then $T$ is a Weiser slice. Our lazy semantics, unlike standard semantics, has the property that it is preserved by slices produced by Hausler’s algorithm with respect to the slicing criterion\(^1\). In addition, the semantics is substitutive (see Definition 17).

4.1 Lazy Semantics

Using a semantics which is not preserved by slicing to prove correctness of slicing algorithms is problematic. Central to slicing is the concept of neededness, defined in Chapter 3 to

\(^1\)A program and its Hausler slices have the same lazy semantics with respect to the slicing criterion
be the set of variables needed by a set of variables $V$ in program $P$. Intuitively, this is the set of variables whose initial value ‘may affect’ the final value of at least one variable $v$ in $V$ after executing $P$. In Section 3.2 we discussed the difficulties encountered when trying to define neededness semantically using standard semantics. Our aim is to define a new semantics which is preserved by Weiser’s slicing algorithm and make the phrase ‘may affect’ semantically precise with respect to the latter semantics.

In our semantics, same as in lazy semantics [15], variables are allowed to have a $\bot$ value, i.e. some variables are mapped to $\bot$ and others to well defined values. Therefore we can have partially defined states. The set of such states is denoted as $\Sigma^\bot$.

$$\Sigma^\bot : I \rightarrow V_\bot.$$  

Where

$$V_\bot = V \cup \bot.$$  

$V_\bot$ is the union of the set of defined values, $V$, and the bottom value, $\bot$.

The ordering on $\Sigma^\bot$ is now a richer ordering than on $\Sigma_\bot$ as used in the standard semantics where all non $\bot$ states were incomparable. For these partially defined states,

$$\sigma_1 \sqsubseteq \sigma_2 \iff \sigma_2(x) = \bot \implies \sigma_1(x) = \bot \ \forall x \in I.$$  

Since variables can be mapped to $\bot$ we now have the possibility that evaluating an expression in a partially defined state can yield $\bot$. A variable $x$ referenced by an expression $e$ does not necessarily mean it contributes to the evaluation of $e$, for example, the value of the expression $x - x$ is independent of the value of $x$. We define a function, det, which takes an expression $e$ and returns the set of variables referenced by $e$ which contribute to the evaluation of $e$. 


Definition 23 (det):
The function \( \text{det} : E \rightarrow P(I) \) is defined to reflect the variables determining the value of expressions. Given an expression \( e \), we say that a variable \( x \) is in \( \text{det}(e) \) if and only if there exists two states, \( \sigma_1 \) and \( \sigma_2 \) in \( \Sigma \), differing only on the value of \( x \) with \( \mathcal{E}_L[e] \sigma_1 \neq \mathcal{E}_L[e] \sigma_2 \). where, \( \mathcal{E}_L[e] \sigma \) is the lazy value of the expression \( e \) in a state \( \sigma \) defined below.

If \( \text{det}(e) \) contains a variable which has \( \bot \) as a value in \( \sigma \), then the whole expression is evaluated to \( \bot \) in \( \sigma \). Otherwise the lazy value, \( \mathcal{E}_L \), of an expression is the same as its strict value, \( \mathcal{E} \). The meaning of an expression in our lazy semantics is, thus, the function \( \mathcal{E}_L : E \rightarrow \Sigma \bot \hookrightarrow V \).

\[
\text{given by } \mathcal{E}_L[e] \sigma = \begin{cases} 
\bot & \text{if } \exists v \in \text{det}(e) \text{ with } \sigma v = \bot. \\
\mathcal{E}[e] \sigma & \text{otherwise.}
\end{cases}
\]

In Figure 4.1 we show the difference between \( \text{det}(e) \) and the set of variables referenced by some expression, \( \text{Referenced}(e) \). Clearly \( \text{det}(e) \subseteq \text{Referenced}(e) \).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Referenced( (e) )</th>
<th>( \text{det}(e) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e:=x-x+y-z )</td>
<td>( {x, y, z} )</td>
<td>( {y, z} )</td>
</tr>
<tr>
<td>( e:=x+x+y-z )</td>
<td>( {x, y, z} )</td>
<td>( {x, y, z} )</td>
</tr>
</tbody>
</table>

Figure 4.1: Referenced and \( \text{det} \) of an expression.

The lazy meaning of a program is given by the function \( \mathcal{M}_L \), which, as in the case of standard semantics, is a state to state function:

\[
\mathcal{M}_L : P \rightarrow \Sigma \bot \hookrightarrow \Sigma \bot.
\]
4.1 Lazy Semantics

\[
\begin{align*}
x &:= 1; \\
\text{while } (x > 0) &\ x := x + 1; \\
x &:= 5;
\end{align*}
\]

Figure 4.2: Recovering the value of \( x \).

\[
\begin{align*}
\text{if } (z > 0) \\
&\quad \text{then } \\
&\quad \{ \\
&\quad \quad x := 1; \\
&\quad \quad y := 2; \\
&\quad \}
\end{align*}
\]

Figure 4.3: With initial state \( \{x \mapsto 1, y \mapsto 1, z \mapsto \bot\} \), \( x \) has 1 as its final value, whereas \( y \) has \( \bot \).

The lazy semantics of each construct of our simple \( \text{while} \) language defined in Definition 25.

The lazy semantics of \( \text{while} \) loops and if statements is given in terms of the meet operator, \( \sqcap \), which is defined in the following definition.

**Definition 24 (Meet - \( \sqcap \))**

Let \( \sigma_1, \sigma_2, \ldots, \sigma_n \) be \( n \) states in \( \Sigma^\perp \). Then the meet of these states is defined as follows:

\[
\prod_{i=1}^{n} \sigma_i = \begin{cases} \\
\lambda v \cdot \sigma_1(v) & \text{if } \sigma_1(v) = \sigma_i(v) \ \forall \ 1 \leq i \leq n \\
\bot & \text{otherwise.}
\end{cases}
\]

**Definition 25 (Lazy Semantics - \( \mathcal{M}_L \))**:

\[
\mathcal{M}_L : P \rightarrow \Sigma^\perp \rightarrow \Sigma^\perp
\]

**Lazy semantics of the skip statement**
\[ \mathcal{M}_L[\text{skip}] = \lambda \sigma \cdot \sigma \]

As in standard semantics, the meaning of \text{skip} is the identity function on states.

**Lazy semantics of the abort statement**

\[ \mathcal{M}_L[\text{abort}] = \lambda \sigma \cdot \sigma \]

Unlike in standard semantics, the lazy meaning of the \text{abort} statement is the same as lazy meaning of the \text{skip} statement. This is a fundamental difference between lazy and standard semantics. Because of this, successive unfoldings of loops may not be monotonic.

**Lazy semantics of assignment statements**

\[ \mathcal{M}_L[x := e] = \lambda \sigma \cdot \sigma[x := \mathcal{E}_L[e]|\sigma] \]

As in standard semantics, the meaning of an assignment is obtained by updating the state with the new value of the variable assigned to it. In the case of lazy semantics, this value is the lazy value of the corresponding expression. Since in lazy semantics, there are states which map some variables to proper values and other variables to \( \perp \), the assignment rule implies that a variable can ‘recover’ from being undefined as shown in Figure 4.2, where after the loop \( x \) has the value \( \perp \) but it recovers to 5 after the assignment \( x := 5 \).

**Lazy semantics of the sequences of statements**

\[ \mathcal{M}_L[S_1 ; S_2] = \mathcal{M}_L[S_2] \circ \mathcal{M}_L[S_1] \]

As in standard semantics, the lazy meaning of a sequence of statements is simply the composition of the meanings of the individual statements.

**Lazy semantics of if statements**

\[ \mathcal{M}_L[\text{if } (B) \text{ then } S_1 \text{ else } S_2] \]

\[ = \lambda \sigma \cdot \mathcal{E}_L[B]|\sigma \rightarrow \mathcal{M}_L[S_1]|\sigma, \mathcal{M}_L[S_2]|\sigma, \mathcal{M}_L[S_1]|\sigma \cap \mathcal{M}_L[S_2]|\sigma \]

where \( \sigma_1 \cap \sigma_2 \) the meet of \( \sigma_1 \) and \( \sigma_2 \) is defined as \( \lambda i \cdot \sigma_1(i) = \sigma_2(i) \rightarrow \sigma_1(i), \perp \)
The notation \( a \rightarrow b, c, d \) is shorthand for if \( a \) is true return \( b \) otherwise if \( a \) is false return \( c \) otherwise if \( a \) is \( \perp \) return \( d \) and the notation \( a \rightarrow b, c \) is shorthand for if \( a \) is true return \( b \) otherwise return \( c \). If the predicate of an if statement is evaluated to \text{True} or \text{False} the lazy meaning rules are the same as those of the standard semantics. The only difference is when the guard evaluates to bottom. What should the meaning of the if statement be in this case? If a variable \( x \) is assigned different values in the then and else parts its value is \( \perp \). On the other hand, if the value of \( x \) is the same in the then and else parts then this should be its final value even if the guard is \( \perp \), as, in this case, the value of \( x \) does not depend on the guard.

For example given an initial state \( \sigma \{ x \mapsto 1, y \mapsto 1, z \mapsto \perp \} \) in \( \Sigma^\perp \), the value of the if predicate in the program in Figure 4.3 in \( \sigma \) is equal to \( \perp \). However, the value of the variable \( x \) after executing the then branch is the same as when executing the else branch and is equal to 1. In this case the lazy value of the variable \( x \) after executing the program in Figure 4.3 in state \( \sigma \) is equal to 1. Unlike the variable \( x \), the value of the variable \( y \) is different when executing the then branch from when executing else branch, and hence, the final value of the variable \( y \) is \( \perp \).

**Lazy semantics of while loops**

\[
\mathcal{M}_L[\text{while} \ (B) \ S] = \lambda \sigma \cdot \bigwedge_{i=0}^{\infty} (G_i\sigma)
\]

where

\[
G_i\sigma = \bigwedge_{n=i}^{\infty} \mathcal{M}_L[\mathcal{W}_n(B, S)]\sigma.
\]

\( G_i \) is just the meet of the lazy meaning of the \( n \)th unfoldings, \( \mathcal{W}_n \), for all \( n \geq i \). The unfolding is given in Definition 20, where the meet is defined in Definition 24.

Given a state \( \sigma \) and a variable \( x \) the final lazy value of \( x \) after executing a while loop starting in state \( \sigma \) is the limit of all the values of \( x \) after executing each of the unfoldings.
If the limit does not exist, then we define the final lazy value to be \( \bot \). Here we mean the limit with respect to a discrete metric i.e., for the limit to exist, there must exist an \( N \in \mathbb{N} \) such that all unfoldings greater than \( N \) give the same value for \( x \) in \( \sigma \). If this is the case we say the value of \( x \) stabilises after \( N \) unfoldings. The lazy meaning of while loop is thus the limit of the meet of the lazy meaning of all its corresponding unfoldings:

Although the \( \mathcal{M}_L[\mathcal{W}_n(B, S)] \) is not monotonic, i.e. \( \mathcal{M}_L[\mathcal{W}_n(B, S)] \) is not necessarily less defined than \( \mathcal{M}_L[\mathcal{W}_{n+1}(B, S)] \), clearly \( G_i \subseteq G_{i+1} \), hence the least upper bound of the \( G_i \) exists.

In states where the while loop does not terminate, if the value of the variable stabilises after \( i \) unfoldings for some \( i \geq 0 \), then its meaning will be the stabilised value. Otherwise, its value is just \( \bot \). For example, given the infinite loop in the program in Figure 4.4 the value of the variable \( x \) stabilises to 1 after the first unfolding whereas the value of the variable \( y \) never stabilises. In this case, the lazy values of \( x \) and \( y \) are 1 and \( \bot \) respectively.

\[
\text{while (True)}
\begin{align*}
&\{
&\quad x:=1; \\
&\quad y:=y+1;
&\}
\]

Figure 4.4: The lazy value of \( x \) is 1 and of \( y \) is \( \bot \).

In states where the predicate of the while evaluates to \( \bot \), if the value of a variable is the same and equal to \( v \) for all the unfoldings, after executing zero or more unfoldings, then its value is just \( v \). And if otherwise the variable is evaluated to \( \bot \).

For example, after executing the first infinite loop in the program in Figure 4.5, the value of the variable \( y \) is undefined and therefore, the condition of the second while loop is
z:=1;
x:=1;
while (True)
{
    if(y>0)
        then y:=-1;
    else y:=1;
}
while (y>0)
{
    x:=1;
    z:=2;
}

Figure 4.5: x is evaluated to 1 where the final value of z is ⊥.

undefined. However, the value of the variable x does not change and is equal to 1 after executing the body of the loop zero, a finite or infinite number of times. In this case the value of the variable x after executing the second loop is equal to 1. Unlike, the variable x, the variable z has a different value when the body of the while is not executed at all, which is 1, from its value when the body is executed, which is equal to 2. In this case the final value of the variable z is equal to ⊥.

The example in Figure 4.5 illustrates the difference of our semantics with both the lazy semantics of Cartwright and Felleisen [15] and the transfinite semantics by Giacobazzi and Mastroeni [33]. The final value of the variable x, in both these semantics, when executing the program in Figure 4.5 is ⊥.
4.1 Lazy Semantics

4.1.1 Properties of the Lazy Semantics

An important property of our lazy semantics is that for terminating programs, it agrees with the standard semantics.

**Theorem 4.1.1:** Let $P$ be a program and $\sigma$ be a state in $\Sigma$, then,

$$\mathcal{M}[P]\sigma \neq \bot \implies \mathcal{M}_L[P]\sigma = \mathcal{M}[P]\sigma.$$

**Proof.** This is proved by structural induction over the language being considered, as follows.

**skip Statement**

Trivial as $\mathcal{M}_L[\text{skip}]\sigma = \sigma = \mathcal{M}[\text{skip}]\sigma$ for all $\sigma$ in $\Sigma$.

**abort Statement**

The result is vacuously true as, $\mathcal{M}[\text{abort}]\sigma = \bot$ for all $\sigma$ in $\Sigma$.

**Assignment Statements**

Trivial, $\mathcal{E}_L[e]\sigma = \mathcal{E}[e]\sigma$, $\sigma \in \Sigma$.

**Conditional Statements**

Induction hypothesis: Assume that for all $\sigma$ in $\Sigma$ and for $i = 1, 2$ if $\mathcal{M}[S_i]\sigma \neq \bot$ then $\mathcal{M}_L[S_i]\sigma = \mathcal{M}[S_i]\sigma$.

We need to show that for all $\sigma$ in $\Sigma$, if $\mathcal{M}[\text{if } (B) \text{ then } S_1 \text{ else } S_2]\sigma \neq \bot$ then $\mathcal{M}_L[\text{if } (B) \text{ then } S_1 \text{ else } S_2]\sigma = \mathcal{M}[\text{if } (B) \text{ then } S_1 \text{ else } S_2]\sigma$.

Let $\sigma$ be a state in $\Sigma$ with $\mathcal{M}[\text{if } (B) \text{ then } S_1 \text{ else } S_2]\sigma \neq \bot$. Then it follows that $\mathcal{E}_L[B]\sigma = \mathcal{E}[B]\sigma \neq \bot$ as $\sigma$ is a state in $\Sigma$. If $\mathcal{E}_L[B]\sigma = \text{true}$, then $\mathcal{M}[\text{if } (B) \text{ then } S_1 \text{ else } S_2]\sigma$ is reduced to just $\mathcal{M}[S_1]\sigma$ and $\mathcal{M}_L[\text{if } (B) \text{ then } S_1 \text{ else } S_2]\sigma$
is reduced to just $M_L[S_1]\sigma$. And the result follows immediately from the induction hypothesis. Similarly, if $E_L[B]\sigma = \text{False}$ as $M[\text{if } (B) \text{ then } S_1 \text{ else } S_2]\sigma$ is reduced to just $M[S_2]\sigma$ and $M_L[\text{if } (B) \text{ then } S_1 \text{ else } S_2]\sigma$ is reduced to just $M_L[S_2]\sigma$.

**Sequences**

**Induction hypothesis:** Assume for all $\sigma$ in $\Sigma$ and for $i = 1, 2$ if $M[S_i]\sigma \neq \bot$ then $M_L[S_i]\sigma = M[S_i]\sigma$.

We must show that for all $\sigma$ in $\Sigma$, if $M[S_1; S_2]\sigma \neq \bot$ then

$$M_L[S_1; S_2]\sigma = M[S_1; S_2]\sigma.$$  

Let $\sigma \in \Sigma$ with $M[S_1; S_2]\sigma \neq \bot$. Hence, $M[S_2](M[S_1]\sigma) \neq \bot$ and $M[S_1]\sigma \neq \bot$. The result follows immediately by application of the semantics rule for sequences and the induction hypothesis:

$$M_L[S_1; S_2]\sigma = M_L[S_2](M_L[S_1]\sigma) \quad \text{(by definition)}$$
$$= M[S_2](M[S_1]\sigma) \quad \text{(induction hypothesis)}$$
$$= M[S_1; S_2]\sigma.$$  

**while Loops**

**Induction hypothesis:** for all $\sigma$ in $\Sigma$, if $M[S]\sigma \neq \bot$ then $M_L[S]\sigma = M[S]\sigma$.

Show that if $M[\text{while } (B) \ S]\sigma \neq \bot$, then $M_L[\text{while } (B) \ S]\sigma = M[\text{while } (B) \ S]\sigma$.

Let $\sigma$ be a state in $\Sigma$, such that $\text{while } (B) \ S$ terminates on $\sigma$. Let’s say $\text{while } (B) \ S$ terminates after $n$ iterations, then $M[\text{while } (B) \ S]\sigma = M[\mathcal{W}_i(B, S)]\sigma$ for all $i \geq n$. Thus using the definition of the lazy meaning of while loops, it suffices to show that for all $i \geq 0$, if $M[\mathcal{W}_i(B, S)]\sigma \neq \bot$ then $M_L[\mathcal{W}_i(B, S)]\sigma = M[\mathcal{W}_i(B, S)]\sigma$. We show this by induction on $i$. The base case is vacuously true as $M[\mathcal{W}_0(B, S)]\sigma = \bot$. We now assume that the result holds for $i^{th}$ unfolding: $\forall \sigma \in \Sigma$, if $M[\mathcal{W}_i(B, S)]\sigma \neq \bot$ then $M_L[\mathcal{W}_i(B, S)]\sigma = M[\mathcal{W}_i(B, S)]\sigma$. Let $\sigma$ be a state in $\Sigma$, with $M[\mathcal{W}_{i+1}(B, S)]\sigma \neq \bot$. We must show that $M_L[\mathcal{W}_{i+1}(B, S)]\sigma = M[\mathcal{W}_{i+1}(B, S)]\sigma$.  


If $\mathcal{E}[B]_\sigma = \text{False}$, then $\mathcal{M}_L[\mathcal{W}_{i+1}(B, S)]_\sigma = \mathcal{M}[\mathcal{W}_{i+1}(B, S)]_\sigma = \sigma$.

If otherwise, $\mathcal{E}[B]_\sigma = \text{True}$, then $\mathcal{M}[\mathcal{W}_{i+1}(B, S)]_\sigma$ is reduced to just $\mathcal{M}[\mathcal{W}_{i}(B, S)](\mathcal{M}[S]_\sigma)$ and $\mathcal{M}_L[\mathcal{W}_{i+1}(B, S)]_\sigma$ is reduced to just $\mathcal{M}_L[\mathcal{W}_{i}(B, S)](\mathcal{M}_L[S]_\sigma)$. And the result follows immediately by application of the induction hypothesis on $S$ and $i$. Thus completing the proof. □

As in the case of the standard semantics, the lazy meaning of while $(B) S$, is the same as the lazy meaning of if $(B)$ then $S$; while $(B) S$. This is the content of the next lemma.

**Lemma 4.1.2:**

$$\mathcal{M}_L[\text{while } (B) S] = \mathcal{M}_L[\text{if } (B) \text{ then } S; \text{ while } (B) S].$$

**Proof.** Let $\sigma$ in $\Sigma^\perp$, By Definition 25 there is an integer $n_0 \geq 0$ such that for all $n \geq n_0$

$$\mathcal{M}_L[\text{while } (B) S]_\sigma = \bigcap_{i=n}^{\infty} \mathcal{M}_L[\mathcal{W}_i(B, S)]_\sigma$$

(4.1)

$$\mathcal{M}_L[\text{while } (B) S]_\sigma \mathcal{M}_L[S]_\sigma = \bigcap_{i=n}^{\infty} \mathcal{M}_L[\mathcal{W}_i(B, S)] \mathcal{M}_L[S]_\sigma.$$

From equation 4.1 it follows that for $n \geq n_0$ we have:
\[ \mathcal{M}_L[\text{while } (B) \ S] \sigma = \prod_{i=n}^{\infty} \mathcal{M}_L[\mathcal{W}_i+1(B, S)] \sigma \]

\[ = \prod_{i=n}^{\infty} \mathcal{M}_L[\text{if } (B) \text{ then } S; \mathcal{W}_i(B, S)] \sigma \quad \text{(by definition)} \]

\[ = \prod_{i=n}^{\infty} (\mathcal{E}_L[B] \sigma \rightarrow \mathcal{M}_L[\mathcal{W}_i(B, S)] \mathcal{M}_L[S] \sigma, \ \sigma, \]

\[ (\mathcal{M}_L[\mathcal{W}_i(B, S)] \mathcal{M}_L[S] \sigma) \cap \sigma) \quad \text{(if rule)} \]

\[ = \mathcal{E}_L[B] \sigma \rightarrow \prod_{i=n}^{\infty} \mathcal{M}_L[\mathcal{W}_i(B, S)] \mathcal{M}_L[S] \sigma, \ \sigma, \]

\[ (\mathcal{M}_L[\text{while } (B) \ S] \mathcal{M}_L[S] \sigma) \cap \sigma) \quad \text{(equation 4.1)} \]

\[ = \mathcal{E}_L[B] \sigma \rightarrow \mathcal{M}_L[\text{while } (B) \ S] \mathcal{M}_L[S] \sigma, \ \sigma, \]

\[ (\mathcal{M}_L[\text{while } (B) \ S] \mathcal{M}_L[S] \sigma) \cap \sigma \quad \text{(sequence rule)} \]

\[ = \mathcal{M}_L[\text{if } (B) \text{ then } S; \text{while } (B) \ S] \sigma \quad \text{(if rule).} \]

Thus completing the proof. \qed

### 4.1.2 Lazy Semantic Discrimination Relation

**Definition 26:** \((x, y) \in \text{LSD}(P)\) if and only if there exists two states \(\sigma_1\) and \(\sigma_2\) which differ only on \(y\) such that:

\[ \mathcal{M}_L[P] \sigma_1 x \neq \mathcal{M}_L[P] \sigma_2 x. \]
Lemma 4.1.3: \( PSSD(P) \subseteq LSD(P) \) for all programs \( P \).

Proof. let \( x, y \) be two variables with \( (x, y) \) in \( PSSD(P) \). We show that \( (x, y) \) is in \( LSD(P) \). By Definition 22, it follows that there exists two states \( \sigma_1 \) and \( \sigma_2 \) which differ only on \( y \) such that:

\[
\bot \neq M[P]_\sigma_1 x \neq M[P]_\sigma_2 x \neq \bot.
\]

We have shown before that the lazy and standard semantics agree for terminating programs (Theorem 4.1.1). Hence,

\[
M_L[P]_\sigma_1 x \neq M_L[P]_\sigma_2 x.
\]

Thus, the result follows by a straightforward application of Definition 26.

4.1.3 Lazy Semantics is Substitutive

Program transformation is a form of program analysis or manipulation. Program transformation alters the syntax of a program while preserving its semantics: We can substitute some parts of a program by their corresponding equivalent and still preserve the semantics of the original program. Therefore, substitutivity (see definition 17) is an important property a semantics should have if it is to be useful to prove correctness of program transformation algorithms slicing [9, 38, 42].

In Chapter 2, we showed that the semantics Cartwright and Felleisen [15] is not substitutive Section 2.7.3. In Section 2.7.4 we showed that the transfinite semantics of Giacobazzi and Mastroeni [33] is not substitutive either. Unlike these semantics our new lazy semantics is substitutive. This is shown in the next theorem.

Theorem 4.1.4 (Our lazy semantics is substitutive):
Let $P$ be a program and $P'$ be a program obtained by replacing a sub-program, $Q$, of $P$ by a $Q'$. Then

$$\mathcal{M}_L[Q] = \mathcal{M}_L[Q'] \implies \mathcal{M}_L[P] = \mathcal{M}_L[P'].$$

Proof. This is proved by structural induction over the language being considered. The result for base case is trivial as \texttt{abort}, \texttt{skip} and assignment statements are atomic statements. For if statements, while loops and sequences the induction step follows by a straightforward application of the lazy meaning rules and the induction hypothesis as their lazy meaning is in terms of their corresponding sub-components. \hfill \square

In this section it has been shown that in all states $\sigma \in \Sigma$ where a program $P$ terminates, its standard and lazy semantics are the same (see Theorem 4.1.1). We also showed that our lazy semantics is substitutive (see Theorem 4.1.4) In the following section we will define the semantics of slicing using lazy semantics.

### 4.2 Defining Slices using Lazy Semantics

We define $P$ and $Q$ to be $V$-lazy equivalent if and only if they have the same lazy semantics with respect to the set of variables $V$.

**Definition 27 (V-lazy equivalence):**

Let $V$ be a set of variables and $P$ and $Q$ be two programs. We say $P$ and $Q$ are $V$-lazy equivalent if and only if for all $\sigma$ in $\Sigma^\bot$ and for all $x$ in $V$, $\mathcal{M}_L[P]\sigma x = \mathcal{M}_L[Q]\sigma x$. This is clearly an equivalence relation.

We write $P \bowtie V Q$ to denote that $P$ and $Q$ are $V$-lazy equivalent.

Figure 4.6 shows two programs, $P_1$ and $P_2$, with the same lazy semantics with respect to


\begin{verbatim}
x:=1;
y:=0;
while (y ≥ 0) {x:=0;}
\end{verbatim}

Program $P_1$.

\begin{verbatim}
x:=0;
\end{verbatim}

Program $P_2$.

Figure 4.6: $P_1$ and $P_2$ have the same lazy semantics w.r.t. the variable $x$.

$x.$ i.e. $P_1\{x\} \approx P_2.$
4.2 Defining Slices using Lazy Semantics

\[ \begin{align*}
x &:= 1; \\
y &:= 0; \\
\text{while } (y = 0) \{ y := y + 1; \} \\
x &:= 0;
\end{align*} \]

A program \( P \).

\[ \begin{align*}
x &:= 0;
\end{align*} \]

End-slice of \( P \) w.r.t. \( x \).

Figure 4.7: \( P \) and its end-slice have the same lazy semantics w.r.t. \( x \).

**Definition 28 (Weiser’s semantic definition of a slice):**

Let \( V \) be a set of variables and \( P \) and \( Q \) be two programs. We say that \( Q \) is a slice of \( P \) with respect to \( V \) if and only if, for all \( \sigma \) in \( \Sigma \),

\[ \mathcal{M}[P]_{\sigma} \neq \bot \implies \mathcal{M}[P]_{\sigma x} = \mathcal{M}[Q]_{\sigma x} \quad \forall \ x \in V. \]

The program \( P \) in Figure 4.7 does not terminate. Using Weiser’s semantic definition, the slice of the program \( P \) can be anything. This is a result of the fact that Weiser’s definition does not take into account non-termination.

Unlike Weiser’s standard semantic definition of a slice, Definition 28, page 116, we require a slice to preserve the semantics of the original in all states, not just for ones where the program terminates. We do not wish to allow arbitrary semantics for slices of non-terminating programs since Hausler’s algorithm does not behave in an arbitrary way in these cases. As will be shown in Chapter 5, Hausler’s algorithm preserves lazy semantics given in Definition 25. We now give a new semantic definition of a slice, called a lazy \( V \)-slice, which we prove to be consistent with Weiser’s one.

**Definition 29 (Lazy \( V \)-Slice):**

Let \( P \) and \( P_1 \) be two programs, and \( V \) be a set of variables. We say \( P_1 \) is a lazy \( V \)-slice of
a program $P$ if and only if

$$P \overset{\mathcal{V}}{\sim} P_1, \text{ and } (\mathcal{M}[P]_\sigma \neq \perp \implies \mathcal{M}[P_1]_\sigma \neq \perp \forall \sigma \in \Sigma).$$

Using our new lazy semantic definition of a slice, it is clear that $x:=0$; is a semantically valid slice of the program $P$ in Figure 4.7 on page 116.

Our definition of a lazy V-slice is one that preserves termination and lazy semantics of the original program projected onto all variables in V. Weiser’s semantics definition of a slice is one that preserves termination and the projected standard semantics onto $V$ only for terminating programs. Our lazy semantics agrees with the standard semantics in all states where the program terminates, (see Theorem 4.1.1, page 109). From this it follows immediately that any slice which satisfies our new definition also satisfies Weiser’s definition semantic definition of slicing.

As a demonstration of the applicability of our lazy semantics, in Chapter 5, Hausler’s slicing algorithm [43] is proved to be correct with respect to Weiser’s definition of a slice. The way we do this is to prove that Hausler’s slicing algorithm [43] is correct with respect to our lazy semantic definition of a slice, which is stronger than Weiser’s one.

A number of lemmas exploring some properties of $\overset{\mathcal{V}}{\sim}$ are given in the following section.

### 4.2.1 Properties of $\overset{\mathcal{V}}{\sim}$

The first lemma is a straightforward consequence of the definitions.

**Lemma 4.2.1:** Given two sets, $V_1 \subseteq V_2$, and two programs, $P$ and $Q$, then,

$$P \overset{V_2}{\sim} Q \implies P \overset{V_1}{\sim} Q.$$
The following lemma shows that if the body of a while loop does not affect a variable \( x \) then neither do any its unfoldings. This lemma will be used later to show that Hausler slices of a while loop preserve lazy semantics.

**Lemma 4.2.2**: Let \( S \) be a program and \( \mathcal{W}_i(B, S) \) be the \( i \)th unfolding of while \((B)\ S\), then

\[
S \begin{array}{c}
\{x\}
\end{array} \text{skip} \implies \mathcal{W}_i(B, S) \begin{array}{c}
\{x\}
\end{array} \text{skip} \quad \forall \ i \geq 0.
\]

*Proof*. We show this by induction on \( i \). For the base case the result is trivial as \( \mathcal{M}_L[\mathcal{W}_0(B, S)] = \mathcal{M}_L[\text{abort}] = \mathcal{M}_L[\text{skip}] \) for all \( \sigma \in \Sigma^+ \). The induction step follows immediately by a straightforward application of the lazy meaning rules and the induction hypothesis. Thus the proof is complete.

The following lemma is a direct consequence of this.

**Lemma 4.2.3**: Let \( S \) be a program and \( \mathcal{W}_i(B, S) \) be the \( i \)th unfolding of while \((B)\ S\), then

\[
\mathcal{M}_L[S] = \mathcal{M}_L[\text{skip}] \implies \mathcal{M}_L[\mathcal{W}_i(B, S)] = \mathcal{M}_L[\text{skip}] \quad \forall \ i \geq 0.
\]

### 4.3 Lazy Neededness

Having failed to define neededness in terms of standard semantics we now turn to its interpretation in lazy semantics. In this section we define neededness in terms of our lazy semantics and show that lazy neededness satisfies the neededness criterion; i.e. it is sub-sequential, subsumes both Hausler-needed and \emph{PSSD}. 
Definition 30 (Lazy Needed: $\mathbb{N}_{\text{lazy}}$):

$\mathbb{N}_{\text{lazy}} : S \times P(I) \rightarrow P(I)$ where $S$ is the set of programs and $I$ is the set of program variables.

A variable $y$ is in $\mathbb{N}_{\text{lazy}}(P, V)$ if and only if there is a variable $x$ in $V$, and two states, $\sigma_1$ and $\sigma_2$ in $\Sigma^+$, differing only on the value of the variable $y$, such that:

$$M_L[p]_1 x \neq M_L[p]_2 x.$$

From the definition it is clear that lazy needed, $\mathbb{N}_{\text{lazy}}$ is lazy semantically discriminating (LSD)). And by Lemma 4.1.3, it follows that $\mathbb{N}_{\text{lazy}}$ is partially discriminating. In this section we show that lazy neededness, $\mathbb{N}_{\text{lazy}}$, defined in Definition 30 is both sub-sequential and is consistent with Hausler's needed. Some intermediate results, exploring some properties of lazy neededness, are now given.

We begin by showing that lazy neededness of a program with respect to a set of variables $V$ is just the union of its lazy neededness with respect to each one of the variables in $V$.

**Lemma 4.3.1:** Given a set of variables $V$ and program $P$ then,

$$\mathbb{N}_{\text{lazy}}(P, V) = \bigcup_{x \in V} \mathbb{N}_{\text{lazy}}(P, \{x\}).$$

**Proof.** This follows immediately from Definition 30. \qed

**Lemma 4.3.2:** Given two sets of variables, $V_1$ and $V_2$, and program $P$ then,

$$V_1 \subseteq V_2 \implies \mathbb{N}_{\text{lazy}}(P, V_1) \subseteq \mathbb{N}_{\text{lazy}}(P, V_2).$$
4.3 Lazy Neededness

Proof. The result follows immediately from Lemma 4.3.1.  

By definition, the lazy needed set of variables of a program \( P \) with respect to a set of variables \( V \) is the set of variables for which the initial value might affect the lazy final value of some of the variables in \( V \) after executing \( P \). Therefore, if two states, \( \sigma_1 \) and \( \sigma_2 \) in \( \Sigma^+ \) agree on all elements in \( \mathbb{N}_{\text{lazy}}(P, V) \) then the meaning of \( P \) in \( \sigma_1 \) and in \( \sigma_2 \) agree on all elements in \( V \). The following lemma shows that our lazy semantics satisfies this property.

Lemma 4.3.3: Given a set of variables \( V \), a program \( P \) and two states, \( \sigma_1 \) and \( \sigma_2 \), differing only on a set \( V_0 \) of elements not in \( \mathbb{N}_{\text{lazy}}(P, V) \), then

\[
\mathcal{M}_L[P]_{\sigma_1} = \mathcal{M}_L[P]_{\sigma_2}, \quad \forall x \in V.
\]

Proof. We show this by contradiction. By Lemma 4.3.1, it suffices to show the result for \( V = \{x\} \). Suppose there exists two states, \( \sigma_1 \) and \( \sigma_2 \), differing only on elements in \( V_0 \), with \( \mathcal{M}_L[P]_{\sigma_1} \neq \mathcal{M}_L[P]_{\sigma_2} \). And choose \( \sigma_1 \) and \( \sigma_2 \) to be the states differing on a minimal set, \( W \subseteq V_0 \), with \( \mathcal{M}_L[P]_{\sigma_1} \neq \mathcal{M}_L[P]_{\sigma_2} \). Clearly \( W \neq \emptyset \), so choose \( y \in W \) and let \( \sigma'_1 = \sigma_1[y \leftarrow \sigma_2(y)] \). By the minimality follows \( \mathcal{M}_L[P]_{\sigma'_1} = \mathcal{M}_L[P]_{\sigma_2} \). Thus, \( \mathcal{M}_L[P]_{\sigma'_1} \neq \mathcal{M}_L[P]_{\sigma_1} \). This contradicts the minimality of \( W \) unless \( W = \{y\} \). In this case, by definition, \( y \in \mathbb{N}_{\text{lazy}}(P, \{x\}) \) which contradicts the hypothesis.  

The example in Figure 4.8 illustrates this property. \( \mathbb{N}_{\text{lazy}}(P, \{x\}) = \{y\} \). The value of the variable \( x \) after executing \( P \) is always equal to 0 in all states where the value of \( y \) is 0. In all other states the value of \( x \) is equal to 1.

Given an expression \( e \), if a variable \( x \) is not in \( \text{det}(e) \) then the initial value of \( x \) does not
4.3 Lazy Neededness

if (y=0)
  then x:=0;
else x:=1;

Figure 4.8: A simple program $P$.

affect the value of the expression $e$. Therefore, the value of $e$ in all states which agree in all elements in $\text{det}(e)$ is the same. The following lemma illustrates this.

Lemma 4.3.4: Let $e$ be an expression, and $\sigma_1$, $\sigma_2$ be two states in $\Sigma^\perp$, differing only on a set $V$ of variables not in $\text{det}(e)$, then $E_L[e]_{\sigma_1} = E_L[e]_{\sigma_2}$.

Proof. This is entirely similar to the proof of Lemma 4.3.3. \qed

Lemma 4.3.5:

Let $V$ be a set of variables, then $N_{\text{lazy}}(\text{skip}, V) = N_{\text{lazy}}(\text{abort}, V) = V$.

Proof. By Lemma 4.3.1, it suffices to show the result for $V = \{x\}$. The result follows immediately from Definition 30 as for all $\sigma$ in $\Sigma^\perp$ we have

$M_L[\text{skip}]\sigma = M_L[\text{abort}]\sigma = \sigma$.

Proof. \qed

As $M_L[x:=e]\sigma x = E_L[e]_{\sigma}$, for any $\sigma$, the next lemma is clear

Lemma 4.3.6: $N_{\text{lazy}}(x:=e, \{x\}) = \text{det}(e)$.

By definition of lazy neededness we expect the following property to be true: Two programs $P_1$ and $P_2$, with the same lazy semantics with respect to a set of variables $V$, have
4.3 Lazy Neededness

\[
x := 1;
\text{while } (y = 0)
\begin{cases}
x := 0;
\end{cases}
\text{if } (y = 0)
\begin{cases}
x := 0;
\text{else } x := 1;
\end{cases}
\]

Program \(P_1\).

Program \(P_2\).

Figure 4.9: \(P_1\) and \(P_2\) have the same lazy semantics w.r.t. the variable \(x\).

the same lazy needed with respect to \(V\). We show this property in the following lemma.

**Lemma 4.3.7:** Let \(V\) be a set of variables and \(P, Q\) be two programs with \(P \overset{V}{\sim} Q\). Then \(N_{\text{lazy}}(P, V) = N_{\text{lazy}}(Q, V)\).

**Proof.** By Lemma 4.3.1, it suffices to show the result for \(V = \{x\}\). Let \(y\) be in \(N_{\text{lazy}}(P, \{x\})\). Then, by definition there exists two states, \(\sigma_1, \sigma_2 \in \Sigma^L\) differing only on the value of \(y\) with \(M_L[P]\sigma_1 x \neq M_L[P]\sigma_2 x\). As \(P \overset{\{x\}}{\sim} Q\), we have \(M_L[Q]\sigma_1 x \neq M_L[Q]\sigma_2 x\); that is \(y \in N_{\text{lazy}}(P, \{x\})\). Hence, \(N_{\text{lazy}}(P, \{x\})\) is a subset of \(N_{\text{lazy}}(Q, \{x\})\). The converse follows on interchanging \(P\) and \(Q\); thus completing the proof. \(\square\)

For example, the two programs, \(P_1\) and \(P_2\), in Figure 4.9 have the same lazy semantics with respect to the variable \(x\). And on both programs the lazy neededness with respect to the variable \(x\) is just \(\{y\}\).

Lazy neededness of a while loop with respect to a set of variables \(V\) is a subset of the union of lazy neededness with respect to \(V\) of all its unfoldings. The program \(P\) in Figure 4.10 shows that converse is not true as:

\[
N_{\text{lazy}}(P, \{x\}) = \{y\} \text{ where } \bigcup_{i=0}^{\infty} N_{\text{lazy}}(W_i(\text{True}, x := y), \{x, y\}).
\]
4.3 Lazy Neededness

![Image](image_url)

**Figure 4.10:** A simple program $P$ with an infinite loop.

This property is stated in the next lemma.

**Lemma 4.3.8:** $\mathbb{N}_{\text{lazy}}(\text{while } (B) \ S, V) \subseteq \bigcup_{i=0}^{\infty} \mathbb{N}_{\text{lazy}}(\mathcal{W}_i(B, S), V)$

**Proof.** By Lemma 4.3.1, it suffices to show the lemma holds for $V = \{x\}$. Let $y$ be a variable not in $\bigcup_{i=0}^{\infty} \mathbb{N}_{\text{lazy}}(\mathcal{W}_i(B, S), \{x\})$. Hence, by Lemma 4.3.3, it follows that for all states $\sigma_1$ and $\sigma_2$ differing only on the value of the variable $y$, and for all $i \geq 0$ we have $\mathcal{M}_L[\mathcal{W}_i(B, S)] \sigma_1 x = \mathcal{M}_L[\mathcal{W}_i(B, S)] \sigma_2 x$. Thus for all $i \geq 0$, we have $\prod_{n=i}^{\infty} \mathcal{M}_L[\mathcal{W}_n(B, S)] \sigma_1 x = \prod_{n=i}^{\infty} \mathcal{M}_L[\mathcal{W}_n(B, S)] \sigma_2 x$. Hence, by definition of the lazy meaning of the while loop, $\mathcal{M}_L[\text{while } (B) \ S] = \bigcup_{i=0}^{\infty} (\prod_{n=i}^{\infty} \mathcal{M}_L[\mathcal{W}_n(B, S)])$, it follows that $\mathcal{M}_L[\text{while } (B) \ S] \sigma_1 x = \mathcal{M}_L[\text{while } (B) \ S] \sigma_2 x$. Therefore, the variable $y$ is not in $\mathbb{N}_{\text{lazy}}(\text{while } (B) \ S, \{x\})$; thus completing the proof. \hfill $\square$

A variable $y$ which does not affect either the value of the predicate of an if statement or the final value of $x$ after executing its True and False parts, won't affect the final value of $x$ after executing the if statement. This is stated in the next lemma.

**Lemma 4.3.9:** $\mathbb{N}_{\text{lazy}}(\text{if } (B) \ S_1 \text{ else } S_2, V) \subseteq \mathbb{N}_{\text{lazy}}(S_1, V) \cup \mathbb{N}_{\text{lazy}}(S_2, V) \cup \text{det}(B)$.

**Proof.** By Lemma 4.3.1, it suffices to show that the result holds for $V = \{x\}$. Let $y$ be a variable not in $\text{det}(B) \cup \mathbb{N}_{\text{lazy}}(S_1, \{x\}) \cup \mathbb{N}_{\text{lazy}}(S_2, \{x\})$. By Lemma 4.3.4 and Lemma 4.3.3 it follows that for all states $\sigma_1$, $\sigma_2$ differing only the value of the variable $y$ we have
\[
\mathcal{E}_L[B]_{\sigma_1} = \mathcal{E}_L[B]_{\sigma_2} \\
\mathcal{M}_L[S_1]_{\sigma_1} x = \mathcal{M}_L[S_1]_{\sigma_2} x \\
\mathcal{M}_L[S_2]_{\sigma_1} x = \mathcal{M}_L[S_2]_{\sigma_2} x
\]

and the result follows as

\[
\mathcal{M}_L[\text{if } (B) \text{ } S_1 \text{ else } S_2]_{\sigma_i} x = \mathcal{E}_L[S]_{\sigma_i} \rightarrow \mathcal{M}_L[S_1]_{\sigma_i} x, \mathcal{M}_L[S_2]_{\sigma_i} x, \mathcal{M}_L[S_1]_{\sigma_i} x \cap \mathcal{M}_L[S_2]_{\sigma_i} x.
\]

\[
\square
\]

### 4.3.1 Lazy Neededness is Sub-sequential

In the introduction we have explained our intuitive reasons why neededness has to satisfy the *sub-sequentiality* property. The objective of this section is to show that our definition of neededness, lazy neededness, satisfies this property.

**Theorem 4.3.10**: Let \( P_1 \) and \( P_2 \) be two programs and \( V \) a set of variables, then

\[
\mathbb{N}_{\text{lazy}}(P_1; P_2, V) \subseteq \mathbb{N}_{\text{lazy}}(P_1, \mathbb{N}_{\text{lazy}}(P_2, V)).
\]

**Proof.** By Lemma 4.3.1, it suffices to show the theorem holds when \( V = \{x\} \). Let \( \sigma_1 \) and \( \sigma_2 \) be two states differing only on the value of \( y \), which is not in \( \mathbb{N}_{\text{lazy}}(P_1, \mathbb{N}_{\text{lazy}}(P_2, \{x\})) \). Hence, by Lemma 4.3.3, it follows \( \mathcal{M}_L[P_1]_{\sigma_1} \) and \( \mathcal{M}_L[P_1]_{\sigma_2} \) agree in all elements in \( \mathbb{N}_{\text{lazy}}(P_2, \{x\}) \). Thus, by Lemma 4.3.3, \( \mathcal{M}_L[P_2]((\mathcal{M}_L[P_1]_{\sigma_1})_x = \mathcal{M}_L[P_2]((\mathcal{M}_L[P_1]_{\sigma_2})_x \). Hence, the variable \( y \) is not in \( \mathbb{N}_{\text{lazy}}(P_1; P_2, \{x\}) \) as \( \mathcal{M}_L[P_1; P_2]_{\sigma} = \mathcal{M}_L[P_2](\mathcal{M}_L[P_1]_{\sigma}) \) for all \( \sigma \) in \( \Sigma^\perp \). Thus, the result follows. \( \square \)

Theorem 4.3.10 shows that lazy neededness satisfies the sub-sequentiality property. In the
following section we show that lazy needed, $N_{lazy}$, is consistent with Hausler-needed.

### 4.3.2 Lazy Neededness is Consistent with Hausler-Neededness

Hausler’s needed is defined in Section 3.1.1, Chapter 3. He uses all data control dependencies to define neededness. Clearly Hausler neededness contains variables which are not needed semantically. Therefore, as we have discussed before, any semantic definition of lazy neededness has to be consistent with Hausler’s one. To show that our lazy semantics satisfies this property we need to show that lazy needed is contained in Hausler-needed. This will be shown in Theorem 4.3.12.

The following lemma is part of the induction step required to show that the slice produced by lazy needed of while loop is contained in its corresponding Hausler one.

**Lemma 4.3.11:** Let $S$ be a program and suppose for all sets of variables $V$ we have $N_{lazy}(S,V) \subseteq N_h(S,V)$. Then for all sets of variables $V$ and for all $i \geq 0$, $N_{lazy}(W_i(B,S),V) \subseteq N_h(\text{while } (B) \ S, V)$.

**Proof.** By definition of Hausler’s algorithm, $N_h(\text{while } (B) \ S, V) = \bigcup_{i=0}^{\infty} \gamma^i(V)$, where,

$$
\gamma^0(V) = V \quad \text{and} \quad \gamma^{n+1}(V) = N_h(\text{if } (B) \ S \text{ else skip}, \gamma^n(V)).
$$

Hence, it suffices to show that for all $i \geq 0$, $N_{lazy}(W_i(B,S),V) \subseteq \gamma^i(V)$.

We show this by induction on $i$. Let $V$ be a set of variables. By Lemma 4.3.5 it follows that $N_{lazy}(\text{abort},V) = V = \gamma^0(V)$. Hence, the result for the base case follows immediately as $W_0(B,S) = \text{abort}$.

We now assume that $N_{lazy}(W_i(B,S),V) \subseteq \gamma^i(V)$. We then need to show that $N_{lazy}(W_{i+1}(B,S),V) \subseteq \gamma^{i+1}(V)$. We have two cases to consider:
If $S_n(S, V) = \text{skip}$. In this case we have $n \geq 0$ we have $\gamma^n(V) = V$. From Lemma 4.2.3 it follows that $\mathcal{W}_n(B, S) \not\sim \text{skip}$, for all $n \geq 0$ and the result follows immediately from Lemma 4.3.7 as $N_{\text{lazy}}(\text{skip}, V) = V$.

If otherwise, then $\gamma^{i+1}(V) = N_h(S, \gamma^i(V)) \cup \gamma^i(V) \cup \text{Referenced}(B)$. And the result follows immediately by a straightforward application of lazy needed rule for if statements and the induction hypothesis:

$$N_{\text{lazy}}(\mathcal{W}_{i+1}(B, S), V)$$

$$= N_{\text{lazy}}(\text{if} (B) S; \mathcal{W}_i(B, S) \text{ else} \text{skip}, V) \quad \text{(by definition)}$$

$$\subseteq N_{\text{lazy}}(S, N_{\text{lazy}}(\mathcal{W}_i(B, S), V)) \cup V \cup \text{det}(B) \quad \text{(Lemma 4.3.9)}$$

$$\subseteq N_{\text{lazy}}(S, \gamma^i(V)) \cup V \cup \text{det}(B) \quad \text{(Lemma 4.3.2 \& ind. hyp.)}$$

$$\subseteq N_h(S, \gamma^i(V)) \cup V \cup \text{Referenced}(B) \quad \text{(hypothesis of the lemma)}$$

$$\subseteq \gamma^{i+1}(V).$$

Thus completing the proof.

\[\square\]

**Theorem 4.3.12 (Lazy needed is Hausler—consistent):**

*Let $P$ be a program and $V$ be a set of variables, then, $N_{\text{lazy}}(P, V) \subseteq N_h(P, V)$.***

*Proof. This is proved by structural induction on the language. Let $V$ be a set of variables.*

**skip statements**

Trivial, by Lemma 4.3.5 $N_{\text{lazy}}(\text{skip}, V) = V = N_h(\text{skip}, V)$.

**abort statements**

Trivial, by Lemma 4.3.5 $N_{\text{lazy}}(\text{abort}, V) = V = N_h(\text{abort}, V)$.

**Assignment statements**

To show that $N_{\text{lazy}}(x:=e, V) \subseteq N_h(x:=e, V)$, two cases need to be considered. If $x$ is not an element of $V$ then $x:=e \not\sim \text{skip}$. By Lemma 4.3.7 and Lemma 4.3.5 we have $N_{\text{lazy}}(x:=e, V) = V$. And the result follows as $N_h(x:=e, V) = V$. Otherwise, $x \in V$, By Lemma 4.3.1 and Lemma 4.3.6, we have $N_{\text{lazy}}(x:=e, V) = (V \setminus \{x\}) \cup \text{det}(e)$. The result
follows immediately as \( N_h(x = e, V) = (V \setminus \{x\}) \cup \text{Referred}(e) \) and \( \text{det}(e) \) is a subset of \( \text{Referred}(e) \).

Sequences

Induction hypothesis: Let \( S_1 \) and \( S_2 \) be two programs with \( N_{\text{lazy}}(S_i, V) \subseteq N_h(S_i, V) \) for all sets of variables \( V \) and for \( i \in \{1, 2\} \). And show that for all sets of variables \( V, N_{\text{lazy}}(S_1; S_2, V) \subseteq N_h(S_1; S_2, V) \). By definition Hausler-needed is sequential i.e. \( N_h(S_1; S_2, V) = N_h(S_1, N_h(S_2, V)) \). Hence, the result follows immediately from Theorem 4.3.10 and Lemma 4.3.2.

if statements

Induction hypothesis: Let \( S_1 \) and \( S_2 \) be two programs with \( N_{\text{lazy}}(S_i, V) \subseteq N_h(S_i, V) \) for all sets of variables \( V \) and for \( i \in \{1, 2\} \). We need to show that for all sets of variables \( V, \ N_{\text{lazy}}(\text{if} (B) \ S_1 \ \text{else} \ S_2, V) \subseteq N_h(\text{if} (B) \ S_1 \ \text{else} \ S_2, V) \).

If \( S_h(\text{if} (B) \ S_1 \ \text{else} \ S_2, V) = \text{skip} \), then by Lemma 5.1.1 and Lemma 4.3.7 we have \( N_{\text{lazy}}(\text{if} (B) \ S_1 \ \text{else} \ S_2, V) = V \). The result follows immediately as in this case we have \( N_h(\text{if} (B) \ S_1 \ \text{else} \ S_2, V) = V \).

Otherwise, \( N_h(\text{if} (B) \ S_1 \ \text{else} \ S_2, V) = N_h(S_1, V) \cup N_h(S_2, V) \cup \text{Referred}(B) \). \( \text{det}(B) \) is contained in \( \text{Referred}(B) \), hence the result follows immediately by application of the induction hypothesis and Lemma 4.3.9.

while statements

Induction hypothesis: let \( S \) be a program with \( N_{\text{lazy}}(S, V) \subseteq N_h(S, V) \) for all sets of variables \( V \). We need to show that \( N_{\text{lazy}}(\text{while} (B) \ S, V) \subseteq N_h(\text{while} (B) \ S, V) \) for all sets of variables \( V \). The result follows immediately from Lemma 4.3.8 and Lemma 4.3.11. Thus completing the proof.
In this chapter, we have defined the lazy semantics of a simple imperative programming language and proved that in cases where a program terminates, its lazy and standard semantics are the same. We also have shown that our lazy semantics is substitutive. In section 4.2, we define a new semantic definition of slicing, called lazy semantic definition of a slice, with respect to our lazy semantics. Our definition is stronger and consistent with Weiser’s one. In Section 4.3 we have defined a semantic definition of neededness, lazy semantics definition of Neededness, in terms of our lazy semantics. We also showed that our lazy semantics definition of Neededness satisfies the neededness criterion.

As a demonstration of the applicability of our lazy semantics, in the next chapter a correctness proof of Hausler’s slicing algorithm is given.
Chapter 5

Correctness Proof of Hausler’s Slicing Algorithm

Using standard semantics to prove correctness of slicing algorithms proves to be very difficult due to non-termination. In lazy semantics the non-termination is taken into account and all the difficulties that arise in standard semantics disappear. In our lazy semantics definition of a slice, for a slice to be valid, it has to preserve lazy semantics and termination of the original program.

5.1 Lazy Semantics is Preserved by Hausler’s Algorithm

In this section we will show that slices produced by Hausler’s algorithm preserve the lazy semantics. Some properties of the slices produced by Hausler’s algorithm are first given in the form of lemmas.

Lemma 5.1.1: Let $S$ be a program and $V$ be a set of variables, then

$$S_{\downarrow}(S, V) = \text{skip} \implies S \not\sim \text{skip}.$$ 

Proof. This is can be achieved by structural induction over the language.
5.1 Lazy Semantics is Preserved by Hausler’s Algorithm

**skip statement**

The result is trivial, $\sim$ is an equivalent relation.

**abort statement**

The result is trivial, $\mathcal{M}_L[\text{abort}] = \mathcal{M}_L[\text{skip}]$.

**Assignment statements**

$S_h(x := e, V) = \text{skip}$, implies, by Hausler algorithm, that $x \not\in V$. And the result follows immediately as $\mathcal{M}_L[x := e] \sigma = \sigma[x \leftarrow e][\sigma]$.

**Sequences**

Induction hypothesis: Let $S_1$ and $S_2$ be two programs. Assume for all sets of variables $V$ and for $i \in \{1, 2\}$, if $S_h(S_i, V) = \text{skip}$ then $S_i \sim \text{skip}$. We need to show that the result holds for $S_1; S_2$.

Let $V$ be a set of variables with $S_h(S_1; S_2, V) = \text{skip}$. In this case Hausler’s algorithm implies that: $S_h(S_2, V) = \text{skip}$ and $S_h(S_1, N_h(S_2, V)) = \text{skip}$. By application of the induction hypothesis on $S_2$, we have $S_2 \sim \text{skip}$. By Lemma 4.3.7 it follows $N_{\text{lazy}}(S_2, V) = V$ and by Theorem 4.3.12 we have $N_{\text{lazy}}(S_2, V) \subseteq N_h(S_2, V)$. By application of the induction hypothesis on $S_1$, it follows that $S_1 \sim N_{\text{lazy}}(S_1, V)$ skip. Hence, the result follows immediately as $V \subseteq N_h(S_2, V)$ and for all $\sigma$ in $\Sigma^+$, as $\mathcal{M}_L[S_1; S_2] \sigma = \mathcal{M}_L[S_2]\mathcal{M}_L[S_1] \sigma$.

**if statements**

Induction hypothesis: Let $S_1$ and $S_2$ be two programs. We assume that for all sets of variables $V$ and for $i \in \{1, 2\}$, if $S_h(S_i, V) = \text{skip}$ then $S_i \sim \text{skip}$. Show that for all sets of variables $V$ if $S_h(\text{if } (B) \ S_1 \text{ else } S_2, V) = \text{skip}$ then if $\ (B) \ S_1 \text{ else } S_2 \sim \text{skip}$. Let $V$ be a set of variables with $S_h(\text{if } (B) \ S_1 \text{ else } S_2, V) = \text{skip}$. Then, by definition of Hausler’s algorithm we have $S_h(S_1, V) = S_h(S_2, V) = \text{skip}$. Hence, by induction hypothesis, $S_1 \sim S_2 \sim \text{skip}$. The result follows immediately as for all $\sigma$ in $\Sigma^+$ $\mathcal{M}_L[\text{if } (B) \ S_1 \text{ else } S_2] \sigma$ is reduced to
just $\mathcal{M}_L[S_1]\sigma$ if the predicate evaluates to true or to just $\mathcal{M}_L[S_2]\sigma$ if it evaluates to false or $\mathcal{M}_L[S_1]\sigma \cap \mathcal{M}_L[S_2]\sigma$ if the predicate evaluates to $\bot$.

**while statements**

Induction hypothesis: Let $S$ be a program. Assume that for all sets of variables $V$ if $S_h(S, V) = \text{skip}$ then $S \Vdash \text{skip}$. We need to show that for all sets of variables $V$,

$$S_h(\text{while } (B) \ S, V) = \text{skip} \implies \text{while } (B) \ S \Vdash \text{skip}.$$ 

Let $V$ be a set of variables with $S_h(\text{while } (B) \ S, V) = \text{skip}$. In this case, by Hausler’s definition of a slice, we have $S_h(S, V) = \text{skip}$. By application of the induction hypothesis it follows that $S \Vdash \text{skip}$. Hence, by Lemma 4.2.2 it follows that $W_i(B, S) \Vdash \text{skip}$, for all $i \geq 0$. Thus the result follows immediately as for all $\sigma$ in $\Sigma^\bot$,

$$\mathcal{M}_L[\text{while } (B) \ S]\sigma = \bigcup_{i=0}^{\infty} \bigcap_{n=i}^{\infty} \mathcal{M}_L[W_n(B, S)]\sigma.$$ 

\[\square\]

The following lemma is part of the induction step required to show that the slice produced by Hausler’s algorithm of the while loop with respect to a set of variables of interest $V$ preserves the lazy semantics of the original program with respect to $V$.

**Lemma 5.1.2:** Let $S$ be a program and $B$ be a boolean expression, suppose that for all sets of variables $V$ we have $S_h(S, V) \Vdash S$. Then for all sets of variables $V$ and for all $i \geq 0$, we have $W_i(B, S) \Vdash W_i(B, T)$ where $T = S_h(S, N_h(\text{while } (B) \ S, V))$.

**Proof.** We show this by induction on $i$. Let $V$ be a set of variables and $T$ be a program with $T = S_h(S, N_h(\text{while } (B) \ S, V))$. The base case is trivial as the zeroth unfolding is just abort. Hence, $\mathcal{M}_L[W_0(B, S)] = \mathcal{M}_L[\text{abort}] = \mathcal{M}_L[W_0(B, T)]$. We now assume that $W_i(B, S) \Vdash W_i(B, T)$ and show that $W_{i+1}(B, S) \Vdash W_{i+1}(B, T)$. 


5.1 Lazy Semantics is Preserved by Hausler’s Algorithm

By hypothesis we have $T^{N_h}_\text{while} (B) \sim (S, V) S$. Hence, by Lemma 4.3.11 it follows that $T^{N_{\text{aux}}}_i (B, S, V) S$ for all $i \geq 0$. Finally, by a straightforward application of Lemma 4.3.3, lazy meaning rules and the induction hypothesis the induction step follows immediately, thus completing the proof. $\square$

We now show that Hausler’s slicing algorithm preserves lazy semantics.

**Theorem 5.1.3:** Let $P$ be a program and $V$ be a set of variables, then,

$$S_h(P, V) \sim P.$$

**Proof.** We prove this by structural induction. Let $V$ be a set of variables.

**skip statement**

Trivial as $S_h(\text{skip}, V) = \text{skip}$ and $\sim$ is an equivalence relation.

**abort statement**

Trivial as $S_h(\text{abort}, V) = \text{skip}$ and $\mathcal{M}_L[\text{abort}] = \mathcal{M}_L[\text{skip}]$.

**Assignment statements**

We need to prove that $S_h(x := e, V) \sim x := e$. we have two cases to consider. If $x \in V$, then $S_h(x := e) = x := e$ and the result follows immediately as $\sim$ is an equivalence relation. Otherwise, $S_h(x := e, V) = \text{skip}$ and the result follows immediately from Lemma 5.1.1.

**Sequence statements**

We assume that for all sets of variables $V$ and for $i = 1, 2$ we have $S_h(P_i, V) \sim P_i$. We need to show that for all sets of variables $V$, $S_h(P_1; P_2, V) \sim P_1; P_2$. Hausler’s rule of slicing sequences implies $S_h(P_1; P_2, V) = S_h(P_1, N_h(P_2, V)) ; S_h(P_2, V)$. Let $\sigma$ be a state in $\Sigma$, $\sigma' = \mathcal{M}_L[S_h(P_1, N_h(P_2, V))]\sigma$ and $V$ be a set of variables. By applying lazy meaning
rules for sequences we only have to show that for all \( x \) in \( V \),

\[
\mathcal{M}_L[\mathcal{S}_h(P_2, V)] \sigma' x = \mathcal{M}_L[P_2](\mathcal{M}_L[P_1] \sigma) x.
\]

By applying the induction hypothesis on \( P_1 \) it follows that \( \sigma' \) and \( \mathcal{M}_L[P_1] \sigma \) agree in all elements in \( \mathbb{N}_h(P_2, V) \), i.e. by Theorem 4.3.12 they agree in all elements in \( \mathbb{N}_{lazy}(P_2, V) \). Hence, by Lemma 4.3.3 it follows that for all \( x \in V \), \( \mathcal{M}_L[P_2] \sigma' x = \mathcal{M}_L[P_2](\mathcal{M}_L[P_1] \sigma) x \). And by induction hypothesis on \( P_2 \) we have \( \mathcal{M}_L[\mathcal{S}_h(P_2, V)] \sigma' x = \mathcal{M}_L[P_2] \sigma' x \) for all \( x \) in \( V \). Thus the result follows.

**if statements**

We assume that for all sets of variables \( V \) and for \( i = 1, 2 \) we have \( \mathcal{S}_h(S_i, V) \overset{\nu}{\sim} S_i \). We need to show that for all sets of variables \( V \), we have

\[
\mathcal{S}_h(\text{if } (B) \ S_1 \text{ else } S_2, \ V) \overset{\nu}{\sim} \text{if } (B) \ S_1 \text{ else } S_2.
\]

If \( \mathcal{S}_h(\text{if } (B) \ S_1 \text{ else } S_2, \ V) = \text{skip} \), then the result follows immediately from Lemma 5.1.1.

If otherwise, then \( \mathcal{S}_h(\text{if } (B) \ S_1 \text{ else } S_2, \ V) = \text{if } (B) \ \mathcal{S}_h(S_1, V) \text{ else } \mathcal{S}_h(S_2, V) \).

In this case for all \( \sigma \in \Sigma^\dagger \), if \( \mathcal{E}_L[B] \sigma \) was evaluated to true then the meaning of the if statement would be reduced to just \( \mathcal{M}_L[S_1] \sigma \) and the lazy meaning of its slice with respect to \( V \) is reduced to \( \mathcal{M}_L[\mathcal{S}_h(S_1, V)] \sigma \). The result follows immediately from the induction hypothesis. Similarly, if the predicate is evaluated to false. If the predicate is evaluated to \( \bot \), then the meaning of the if statement is reduced to just \( \mathcal{M}_L[S_1] \sigma \cap \mathcal{M}_L[S_2] \sigma \) and the lazy meaning of its slice with respect to \( V \) is reduced to \( \mathcal{M}_L[\mathcal{S}_h(S_1, V)] \sigma \cap \mathcal{M}_L[\mathcal{S}_h(S_2, V)] \sigma \). Thus, by application of the induction hypothesis, the result follows.

**while statements**

Induction hypothesis: we assume for all sets of variables \( V \) we have \( \mathcal{S}_h(S, V) \overset{\nu}{\sim} S \). We
need to show that for all sets of variables $V$, while $(B) S \sim \mathbb{S}_h(\text{while } (B) S, V)$. Let $V$ be
a set of variables, then if $\mathbb{S}_h(\text{while } (B) S, V) = \text{skip}$. In this case the result follows im-
mEDIATELY from Lemma 5.1.1. If otherwise, then $\mathbb{S}_h(\text{while } (B) S, V) = \text{while } (B) T$, where
$T = \mathbb{S}_h(S, \mathbb{N}_h(\text{while } (B) S, V))$. From Lemma 5.1.2 it follows that for all $i \geq 0$ we have
$\mathcal{W}_i(B, S) \sim \mathcal{W}_i(B, T)$. Thus the result follows immediately as:

$$\mathcal{M}_L[\text{while } (B) S] = \bigcup_{i=0}^{\infty} (\bigcap_{n=i}^{\infty} \mathcal{M}_L[\mathcal{W}_n(B, S)])$$

$$\mathcal{M}_L[\text{while } (B) T] = \bigcup_{i=0}^{\infty} (\bigcap_{n=i}^{\infty} \mathcal{M}_L[\mathcal{W}_n(B, T)])$$.

Thus completing the proof.

Theorem 5.1.3 shows that slices produced by Hausler’s algorithm preserve lazy semantics. Therefore, Hausler’s slices satisfy the first condition of our lazy semantics definition of slices. To prove correctness of Hausler’s algorithm we only need to show that Hausler’s slices preserve termination; if the original program terminates then its slice with respect to a set of variables $V$ will terminate. This will be shown in Section 5.3. Before doing this we need to define semantically the set of variables that might affect the termination of a program. In the following section we define semantically the meaning of termination neededness and and investigate some of its properties.

### 5.2 Termination Neededness

In program slicing, a slice needs to preserve termination. The set of variables for which the initial values may affect only the termination of a program $P$ is of interest. We write this set as $\mathbb{N}(P, \bot)$. In Figure 5.1 the initial value of the variable $y$ affects the termination of the program $P$. In all states with a negative value of $y$ the program does not terminate and in all other states the program $P$ terminates.
5.2 Termination Neededness

\[ x := 1; \]
\[ \text{while } (y < 0) \ x := 1; \]

Figure 5.1: A simple program $P$ with $\mathbb{N}(P, \bot) = \{y\}$.

The main object of this section is to show that the slices produced by Hausler’s algorithm preserve termination. In order to show this, we first need to show that the set of variables affecting the termination of a slice produced by Hauser’s algorithm is a contained in Hausler neededness. We first start by defining formally what is meant by a variable affecting the termination of a program $P$.

**Definition 31:** A variable $x$ is in $\mathbb{N}(p, \bot)$ if and only if there exist two states $\sigma_1$ and $\sigma_2$ differing only on the value of $x$ such that:

\[ \mathcal{M}[P]\sigma_1 = \bot \quad \text{and} \quad \mathcal{M}[P]\sigma_2 \neq \bot \]

For example, let $\sigma_1$ and $\sigma_2$ be two states differing only on the value of the variable $y$, with $\sigma_1(y) = 1$ and $\sigma_2(y) = 0$. In the state $\sigma_1$, the program $P$ in Figure 5.1 will terminate, however it won’t terminate in the state $\sigma_2$. Hence, $y$ is in $\mathbb{N}(P, \bot)$.

If a program $P$ terminates on a state $\sigma$ then $P$ terminates on all states which agree with $\sigma$ in all elements in $\mathbb{N}(P, \bot)$. Similarly if $P$ does not terminate in $\sigma$ then it does not terminate in all states which agree with $\sigma$ on all elements in $\mathbb{N}(P, \bot)$. This property is an important step in showing that Hausler’s slicing algorithm preserves termination. We state this property in the following lemma.
Lemma 5.2.1: Given a program $P$ and $\sigma_1$, $\sigma_2$ two states in $\Sigma$, differing only on a set $V$ of elements not in $N(P, \bot)$, then, $\mathcal{M}[P]_{\sigma_1} = \bot \iff \mathcal{M}[P]_{\sigma_2} = \bot$.

Proof. By contradiction. The proof can be constructed in exactly similar way to the proof of Lemma 4.3.3. We chose two states, $\sigma_1$ and $\sigma_2$, differing on a minimal set $W \subseteq V$, such that $\mathcal{M}[P]_{\sigma_1} = \bot$ and $\mathcal{M}[P]_{\sigma_2} \neq \bot$. And in a similar way we show that $W$ must contain some variables in $N(P, \bot)$. □

The following Lemma is part of the induction step required later to show that for all sets of variables $V$, the set of variables that can affect the termination of $S_h(\text{if } (B) \text{ then } S_1 \text{ else } S_2, V)$ is always contained in $N_h(\text{if } (B) \text{ then } S_1 \text{ else } S_2, V)$.

Lemma 5.2.2: Let $S_1$ and $S_2$ be two programs and $B$ be a boolean expression. Suppose for all sets of variables $V$, $N(S_h(S_i, V), \bot) \subseteq N_h(S_i, V)$ for $i \in \{1, 2\}$. Then, for all sets of variables $V$ we have:

$$N(\text{if } (B) S_h(S_1, V) \text{ else } S_h(S_2, V), \bot) \subseteq N_h(S_1, V) \cup N_h(S_2, V) \cup \text{det}(B).$$

Proof. Let $\sigma_1$ be a state in $\Sigma$ with $\mathcal{M}[\text{if } (B) S_h(S_1, V) \text{ else } S_h(S_2, V)]_{\sigma_1} \neq \bot$. Let $\sigma_2$ be a state in $\Sigma$ differing from $\sigma_1$ only on the value of a variable $y$ not in $\text{det}(B) \cup N_h(S_1, V) \cup N_h(S_2, V)$.

From Lemma 4.3.4 it follows that $\mathcal{E}[B]_{\sigma_1} = \mathcal{E}[B]_{\sigma_2}$.

If $\mathcal{E}[B]_{\sigma_1} = \text{true}$ then for $i = 1, 2$, $\mathcal{M}[\text{if } (B) S_h(S_1, V) \text{ else } S_h(S_2, V)]_{\sigma_i}$ is reduced to $\mathcal{M}[S_h(S_1, V)]_{\sigma_i}$. The hypothesis of the Lemma implies that the variable $y$ is not in $N_h(S_h(S_1, V), \bot)$ as $y \notin N_h(S_1, V)$. Therefore, by application of Lemma 5.2.1 it follows that $\mathcal{M}[\text{if } (B) S_h(S_1, V) \text{ else } S_h(S_2, V)]_{\sigma_2} \neq \bot$. Similarly the same conclusion is reached if the predicate evaluates to false as $y \notin N_h(S_2, V)$. Hence, the variable $y$ is not in $N(\text{if } (B) S(S_1, V) \text{ else } S_h(S_2, V), \bot)$. Thus, the result follows. □
The following Lemma will contribute to the induction step required later to show that the set of variables that can affect the termination of a slice of a sequence of statements, $S_1; S_2$, with respect to a set of variables $V$ is contained in Hausler’s needed of $S_1; S_2$ with respect to $V$, $N_h(S_1; S_2, V)$.

**Lemma 5.2.3:** Let $S_1$ and $S_2$ be two programs. Suppose for all sets of variables $V$ and for $i \in \{1, 2\}$ we have $N(S_h(S_i, V), \bot) \subseteq N_h(S_i, V)$. Then, for all sets of variables $V$, $N(S_h(S_1; S_2, V), \bot)$ is a subset of $N_h(S_1, N_h(S_2, V))$.

**Proof.** Let $V$ be a set of variables and $\sigma_1$ be a state in $\Sigma$, with $M[S_h(S_1; S_2, V)]\sigma_1 \neq \bot$. And let $\sigma_2$ be another state in $\Sigma$, differing from $\sigma_1$ only on the value of a variable $y$ not in $N_h(S_1, N_h(S_2, V))$. It suffices to show that $M[S_h(S_1; S_2, V)]\sigma_2 \neq \bot$.

By definition $M[S_h(S_1; S_2, V)]\sigma_1 = M[S_h(S_2, V)](M[S_h(S_1, N_h(S_2, V))]\sigma_1) \neq \bot$. This implies that $M[S_h(S_1, N_h(S_2, V))]\sigma_1 \neq \bot$. From the hypothesis of the lemma it follows $M[S_h(S_1, N_h(S_2, V))]\sigma_2 \neq \bot$. Let $\sigma_i' = M[S_h(S_1, N_h(S_2, V))]\sigma_i$ for $i = 1, 2$.

From Theorem 4.3.12 it follows that $y \notin N_{u_{z_y}}(S_1, N_h(S_2, V))$. Hence, by Lemma 4.3.3 for all $x$ in $N_h(S_2, V)$, we have $M_L[S_1]\sigma_1 x = M_L[S_2]\sigma_2 x$. By applying Theorem 5.1.3 and Lemma 4.1.1 consecutively it follows that $\sigma_1'$ and $\sigma_2'$ agree on all in $N_h(S_2, V)$. Hence, by hypothesis of the lemma, they agree in all elements in $N(S_2, V), \bot)$. Hence, by Lemma 5.2.1, it follows that $M[S_h(S_2, V)]\sigma_2' \neq \bot$. Thus, the result follows as $M[S_h(S_1; S_2, V)]\sigma_2 = M[S_h(S_2, V)]\sigma_2' \neq \bot$. \hfill $\square$

**Lemma 5.2.4:** $N(\text{while } (B) \ S, \bot) \subseteq \bigcup_{i=0}^{\infty} N(W_i(B), S, \bot)$.

**Proof.** Let $y$ be a variable in $N(\text{while } (B) \ S, \bot)$. By Definition 31 there exists two states, $\sigma_1$ and $\sigma_2$ in $\Sigma$, differing only on the value of the variable $y$, such that $M[\text{while } (B) \ S]\sigma_1 = \bot$ and $M[\text{while } B] \ S]\sigma_2 \neq \bot$.

If $\text{while } (B) \ S$ does not terminate with the initial state $\sigma_1$, then for all $k \geq 0$ we have
5.2 Termination Neededness

\( \mathcal{M}[\text{while } (B) \ S] \sigma_1 = \mathcal{M}[\mathcal{W}_k(B, S)] \sigma_1 = \bot. \) Where \( \text{while } (B) \ S \) does terminate, say after \( n \) iterations, in which case \( \mathcal{M}[\text{while } (B) \ S] \sigma_2 = \mathcal{M}[\mathcal{W}_k(B, S)] \sigma_2 \) for all \( k \geq n \). Hence for all \( k \geq n \) we have \( \mathcal{M}[\mathcal{W}_k(B, S)] \sigma_1 = \bot \) and \( \mathcal{M}[\mathcal{W}_k(B, S)] \sigma_2 \neq \bot \). Hence, the variable \( y \in N(\mathcal{W}_k(B, S), \bot) \) and the result follows.

We now show that for any simple procedural program \( P \), the set of all variables which may affect the termination of the slice of \( P \) with respect to a set of variables \( V, \mathcal{N}_h(S_h(P, V), \bot) \), is contained in Hausler-needed of \( P \) with respect to \( V, \mathcal{N}_h(P, V) \).

**Lemma 5.2.5:** Let \( P \) be a program and \( V \) be a set of variables, then,

\[
\mathcal{N}(S_h(P, V), \bot) \subseteq \mathcal{N}_h(P, V).
\]

**Proof.** This Lemma is now proved by structural induction over the language being considered.

**skip statement**

Trivial, \( \mathcal{N}(\text{skip}, \bot) = \emptyset \) as \( \mathcal{M}[\text{skip}] \sigma = \sigma \neq \bot \ \forall \sigma \in \Sigma \).

**abort statement**

Trivial, \( \mathcal{N}(\text{abort}, \bot) = \emptyset \) as \( \mathcal{M}[\text{abort}] \sigma = \bot \ \forall \sigma \in \Sigma \).

**Assignment statements**

Trivial, \( \mathcal{N}(S_h(x := e, V), \bot) = \emptyset \), as \( \mathcal{M}[x := e] \sigma \neq \bot \ \forall \sigma \in \Sigma \).

**if statements**

Induction hypothesis: Let \( S_1 \) and \( S_2 \) be two programs with \( \mathcal{N}(S_h(S_i, V), \bot) \subseteq \mathcal{N}_h(S_i, V) \) for all sets of variables \( V \) and for \( i \in \{1, 2\} \). Let \( V \) be set of variables and show that
\[ \mathbb{N}(\mathbb{S}_h(\text{if } (B) \ S_1 \ \text{else } S_2, V), \bot) \subseteq \mathbb{N}_h(\text{if } (B) \ S_1 \ \text{else } S_2, V). \]

If \( \mathbb{S}_h(\text{if } (B) \ S_1 \ \text{else } S_2, V) = \text{skip} \), then the result is trivial as \( \mathbb{N}(\text{skip}, \bot) = \emptyset \).

Otherwise,

\[
\mathbb{S}_h(\text{if } (B) \ S_1 \ \text{else } S_2, V) = \text{if } (B) \ \mathbb{S}_h(S_1, V) \ \text{else } \mathbb{S}_h(S_2, V) \ \text{and,} \]

\[
\mathbb{N}_h(\text{if } (B) \ S_1 \ \text{else } S_2, V) = \mathbb{N}_h(S_1, V) \cup \mathbb{N}_h(S_2, V) \cup \text{Referenced}(B). \]

In which case the result follows immediately by application of the induction hypothesis and Lemma 5.2.2 as \( \text{det}(B) \subseteq \text{Referenced}(B) \).

**Sequences**

Induction hypothesis: let \( S_1 \) and \( S_2 \) be two programs with \( \mathbb{N}(\mathbb{S}_h(S_1, V), \bot) \subseteq \mathbb{N}_h(S_1, V) \) and \( \mathbb{N}(\mathbb{S}_h(S_2, V), \bot) \subseteq \mathbb{N}_h(S_2, V) \) for all sets of variables \( V \).

Let \( V \) be set of variables and show that \( \mathbb{N}(\mathbb{S}_h(S_1; S_2, V), \bot) \subseteq \mathbb{N}_h(S_1; S_2, V) \). This follows immediately by a straightforward application of the induction hypothesis and Lemma 5.2.3 as \( \mathbb{N}_h(S_1; S_2, V) = \mathbb{N}_h(S_1, \mathbb{N}_h(S_2, V)) \).

**while statements**

Induction hypothesis: we assume that for all sets of variables \( V \), we have \( \mathbb{N}(\mathbb{S}_h(S, V), \bot) \subseteq \mathbb{N}_h(S, V) \). Then we show that for all sets of variables \( V \), we have \( \mathbb{N}(\mathbb{S}_h(\text{while } (B) \ S, V), \bot) \subseteq \mathbb{N}_h(\text{while } (B) \ S, V) \).

If \( \mathbb{S}_h(\text{while } (B) \ S, V) = \text{skip} \), the result follows immediately as \( \mathbb{N}(\text{skip}, \bot) = \emptyset \).

Otherwise, \( \mathbb{S}_h(\text{while } (B) \ S, V) = \text{while } (B) \ T \), where \( T = \mathbb{S}_h(S, \mathbb{N}_h(\text{while } (B) \ S, V)) \).

In this case \( \text{det}(B) \subseteq \mathbb{N}_h(\text{while } (B) \ S, V) \). By the induction hypothesis we have \( \mathbb{N}(T, \bot) \subseteq \mathbb{N}_h(\text{while } (B) \ S, V) \) as \( \mathbb{N}_h(S, \mathbb{N}_h(\text{while } (B) \ S, V)) \subseteq \mathbb{N}_h(\text{while } (B) \ S, V) \). By Lemma 5.2.4 it suffices to show that \( \mathbb{N}(\mathbb{W}_i(B, T), \bot) \subseteq \mathbb{N}_h(\text{while } (B) \ S, V) \) for all \( i \geq 0 \). We
show this by induction on \( i \). The base case is trivial as \( \mathbb{N}([W_0(B, T), \perp]) = \emptyset \). We now suppose that \( \mathbb{N}([W_i(B, T), \perp]) \) is a subset of \( \mathbb{N}_h \) (while \( (B, S, V) \)) and show that \( \mathbb{N}([W_{i+1}(B, T), \perp]) \) is a subset of \( \mathbb{N}_h \) (while \( (B, S, V) \)).

Let \( \sigma_1 \) be a state in \( \Sigma \) such that \( \mathcal{M}([W_{i+1}(B, T)] \sigma_1 \neq \perp \) and let \( \sigma_2 \) be a state in \( \Sigma \), differing from \( \sigma_1 \) only on the value of a variable \( y \) which is not in \( \mathbb{N}_h \) (while \( (B, S, V) \)). Hence \( \sigma_1 \) and \( \sigma_2 \) agree in all elements in \( \text{det}(B) \) and in \( \mathbb{N}([W_i(B, T), \perp]) \). Therefore, \( \mathcal{E}[B] \sigma_1 = \mathcal{E}[B] \sigma_2 \).

If \( \mathcal{E}[B] \sigma_1 = \text{True} \) then \( \mathcal{M}([W_{i+1}(B, T)] \sigma_i = \mathcal{M}([W_i(B, T)] \mathcal{M}[T] \sigma_i) \) for \( i = 1, 2 \). From this it follows that \( \mathcal{M}[T] \sigma_1 \neq \perp \), hence \( \mathcal{M}[T] \sigma_2 \neq \perp \) as \( y \) is not in \( \mathbb{N}(T, \perp) \). By application of Theorem \( 5.1.3 \), Theorem \( 4.3.12 \) and Lemma \( 4.3.3 \), consecutively, it follows \( \mathcal{M}[T] \sigma_1 \) and \( \mathcal{M}[T] \sigma_2 \) agree in all elements in \( \mathbb{N}_h \) (while \( (B, S, V) \)). Therefore they agree in all elements in \( \mathbb{N}([W_i(B, T), \perp]) \). Finally, by Lemma \( 5.2.1 \) it follows that \( \mathcal{M}([W_{i+1}(B, T)] \sigma_2 \neq \perp \). If \( \mathcal{E}[B] \sigma_1 = \text{False} \) then \( \mathcal{M}([W_{i+1}(B, T)] \sigma_2 \) is reduced to just \( \sigma_2 \), and hence \( \mathcal{M}([W_{i+1}(B, T)] \sigma_2 \neq \perp \). In both cases, \( \mathcal{M}([W_{i+1}(B, T)] \sigma_2 \neq \perp \). Therefore, \( y \) is not in \( \mathbb{N}([W_{i+1}(B, T), \perp]) \), thus completing the proof.

\[ \square \]

### 5.3 Hausler’s Slicing Algorithm Preserves Termination

Lemma \( 5.2.5 \), shows that all the variables for which the initial value may affect the termination of a slice of a program with respect to a set of variables \( V \), are contained in Hausler’s needed of \( P \) with respect to \( V \). This property is a crucial step toward proving that termination is preserved by Hausler’s slicing algorithm. Some intermediate results are given first.

The following Lemma is part of the induction step required later to show that the termination of sequence of statements is preserved by Hausler’s slicing algorithm.

**Lemma 5.3.1:** Let \( S_1 \) and \( S_2 \) be two programs. Suppose for all states \( \sigma \) in \( \Sigma \) and for all sets of variables \( V \), if \( \mathcal{M}[S_i] \sigma \neq \perp \) then \( \mathcal{M}[S_h(S_i, V)]\sigma \neq \perp \) for \( i = 1, 2 \). Then, for all sets of variables \( V \), \( \mathcal{M}[S_1; S_2] \sigma \neq \perp \implies \mathcal{M}[S_h(S_1; S_2, V)]\sigma \neq \perp \) \( \forall \sigma \in \Sigma \).
5.3 Hausler’s Slicing Algorithm Preserves Termination

Proof. Let \( \sigma \) be a state in \( \Sigma \) and \( V \) be a set of variables. Assume, \( \mathcal{M}[S_1; S_2] \sigma \neq \bot \) and show that \( \mathcal{M}[S_h(S_1; S_2, V)] \sigma \neq \bot \).

\( \mathcal{M}[S_1; S_2] \sigma \neq \bot \) implies that \( \mathcal{M}[S_2](\mathcal{M}[S_1] \sigma) \neq \bot \) and \( \mathcal{M}[S_1] \sigma \neq \bot \). The induction hypothesis implies \( \mathcal{M}[S_h(S_2, V)](\mathcal{M}[S_1] \sigma) \neq \bot \) and \( \mathcal{M}[S_h(S_1, N_h(S_2, V))] \sigma \neq \bot \). By Theorem 5.1.3 and Theorem 4.1.1 it follows that \( \mathcal{M}[S_h(S_1, N_h(S_2, V))] \sigma \) and \( \mathcal{M}[S_1] \sigma \) agree in all elements in \( N_h(S_2, V) \). Hence, they agree in all elements in \( N(S_h(S_2, V), \bot) \)(Lemma 5.2.5). Finally by application of Lemma 5.2.1, it follows that \( \mathcal{M}[S_h(S_2, V)](\mathcal{M}[S_h(S_1, N_h(S_2, V))] \sigma) \neq \bot \). Thus, the result follows immediately as \( S_h(S_1; S_2, V) = S_h(S_1, N_h(S_2, V)); S_h(S_2, V) \). \( \square \)

We now show that slices produced by Hausler’s algorithm preserve termination.

**Theorem 5.3.2:** Let \( P \) be a program and \( V \) be a set of variables, then

\[
\mathcal{M}[P] \sigma \neq \bot \implies \mathcal{M}[S_h(P, V)] \sigma \neq \bot \text{ for all } \sigma \in \Sigma
\]

Proof. This Theorem is now proved by structural induction over the language being considered.

**skip statement**

Trivial as \( S_h(\text{skip}, V) = \text{skip} \) and \( \mathcal{M}[\text{skip}] \sigma = \sigma \neq \bot \text{ for all } \sigma \in \Sigma. \)

**abort statement**

Trivial as \( S_h(\text{abort}, V) = \text{skip}. \)

**Assignment statements**

If \( x \in V \) then \( S_h(x:=e, V) = x:=e. \) If otherwise, \( S_h(x:=e, V) = \text{skip}. \) Thus the result follows.
if statements

Induction hypothesis: Let $S_1$ and $S_2$ be two programs. Assume for all $\sigma$ in $\Sigma$ and for all sets of variables $V$ we have:

$$\mathcal{M}[S_1] \sigma \neq \perp \implies \mathcal{M}[S_h(S_1, V)] \sigma \neq \perp \quad i \in \{1, 2\}.$$ 

We need to show the result holds for $\text{if } (B) \ S_1 \ \text{else } S_2$.

If $S_h(\text{if } (B) \ S_1 \ \text{else } S_2, V) = \text{skip}$. The result follows immediately as for all sets of variables $V$ and for all $\sigma \in \Sigma$, $\mathcal{M}[\text{skip}] \sigma = \sigma \neq \perp$.

Otherwise, $S_h(\text{if } (B) \ S_1 \ \text{else } S_2, V) = \text{if } (B) \ S_h(S_1, V) \ \text{else } S_h(S_2, V)$. Hence, $\mathcal{M}[S_h(\text{if } (B) \ S_1 \ \text{else } S_2, V)] \sigma$ is reduced to just $\mathcal{M}[S_h(S_1, V)] \sigma$ if the predicate is evaluated to true and to $\mathcal{M}[S_h(S_2, V)] \sigma$ otherwise. The result then follows immediately by application of the induction hypothesis.

Sequence of statements

Induction hypothesis: Let $S_1$ and $S_2$ be two programs. Assume for all $\sigma$ in $\Sigma$ and for all sets of variables $V$ we have:

$$\mathcal{M}[S_1] \sigma \neq \perp \implies \mathcal{M}[S_h(S_1, V)] \sigma \neq \perp \quad i \in \{1, 2\}.$$ 

We need show that for all sets of variables $V$ and for all $\sigma \in \Sigma$, if $\mathcal{M}[S_1; S_2] \sigma \neq \perp$ then $\mathcal{M}[S_h(S_1; S_2, V)] \sigma \neq \perp$. This follows immediately by application of the induction hypothesis and Lemma 5.3.1.

while statements

Induction hypothesis: Let $S$ be two programs. Assume for all $\sigma$ in $\Sigma$ and for all sets of variables $V$, if $\mathcal{M}[S] \sigma \neq \perp$ then $\mathcal{M}[S_h(S, V)] \sigma \neq \perp$. 
Let $\sigma$ be a state in $\Sigma$ and $V$ be a set of variables. We suppose $\mathcal{M}[\text{while } (B) \ S] \sigma \neq \bot$ and show that $\mathcal{M}[\text{while } (B) \ S, V] \sigma \neq \bot$.

If $S_h(\text{while } (B) \ S, V) = \text{skip}$. The result is trivial as $\mathcal{M}[\text{skip}] \sigma = \sigma \neq \bot$.

Otherwise, $S_h(\text{while } (B) \ S, V) = \text{while } (B) \ T$ where, $T = S_h(S, N_h(\text{while } (B) \ S, V))$. If while $(B) \ S$ terminates after $n$ iterations in $\sigma$, then,

$$\mathcal{M}[\text{while } (B) \ S] \sigma = \mathcal{M}[W_i(B, S)] \sigma \neq \bot \quad \forall i \geq n.$$  

It suffices to show that $\forall n \geq 0$, if $\mathcal{M}[W_n(B, S)] \sigma \neq \bot$ then $\mathcal{M}[W_n(B, T)] \sigma \neq \bot$. We show this by induction on $n$. The base case is vacuously true as $\mathcal{M}[W_0(B, S)] \sigma = \bot$, for all $\sigma$ in $\Sigma$. We now assume the result holds for $n = i$ and show it holds for $n = i + 1$.

Suppose $\mathcal{M}[W_{i+1}(B, S)] \sigma \neq \bot$. If $E[B] \sigma = \text{False}$ then $\mathcal{M}[W_{i+1}(B, T)] \sigma = \sigma \neq \bot$. Otherwise, if $E[B] \sigma = \text{True}$, then $\mathcal{M}[W_{i+1}(B, S)] \sigma$ is reduced to $\mathcal{M}[W_i(B, S)](\mathcal{M}[S] \sigma)$ and $\mathcal{M}[W_{i+1}(B, T)] \sigma$ is reduced to $\mathcal{M}[W_i(B, T)](\mathcal{M}[T] \sigma)$. Hence,

$$\mathcal{M}[W_i(B, S)](\mathcal{M}[S] \sigma) \neq \bot \text{ and } \mathcal{M}[S] \sigma \neq \bot.$$  

And by application of the induction hypothesis it follows that $\mathcal{M}[W_i(B, T)](\mathcal{M}[S] \sigma) \neq \bot$ and $\mathcal{M}[T] \sigma \neq \bot$. Theorem 5.1.3 and Theorem 4.1.1 imply that $\mathcal{M}[S] \sigma$ and $\mathcal{M}[T] \sigma$ agree in all elements of $N_h(\text{while } (B) \ S, V)$. Hence, they agree in all elements in $N(W_i(B, T), \bot)(\text{Lemma 5.2.5})$. Finally, by Lemma 5.2.1, it follows that

$$\mathcal{M}[W_{i+1}(B, T)] \sigma = \mathcal{M}[W_i(B, T)](\mathcal{M}[T] \sigma) \neq \bot.$$  

This completes the proof. \qed

In Chapter 4 we defined a new semantics for a simple while language, called lazy semantics (see Definition 25, page 104), and gave a new semantic definition of a slice with respect to it (see Definition 29, page 117), where a slice has to preserve both lazy semantics and termination of the original program. In Theorem 5.1.3 we showed that Hausler's slices preserve that lazy semantics, and in Theorem 5.3.2 we showed that slices preserve
termination. Hence, Hausler’s slicing algorithm [43] is proved correct with respect to our lazy semantic definition of a slice. Since our lazy semantic definition of a slice is stronger than Weiser’s standard semantic definition of slicing, this proves that Hausler’s Algorithm [43] is correct with respect to the Weiser’s standard definition too.

In the next chapter we propose an extension of our lazy semantics to handle to programs with procedures and procedure calls. We will only consider parameter-free procedures with non local variables.
Chapter 6
A Denotational Interprocedural Program Slicer

Hausler [43] presents a denotational program slicer for a very simple programming language without procedures. In this chapter we extend Hausler’s approach to a more realistic programming language containing (possibly side-effecting, but non-recursive) functions which can be called both as expressions and as statements. Using the denotational approach, slices can be defined in terms of the abstract syntax of the object language without the need of a control flow graph [88] or similar intermediate structure [73, 77, 34].

The algorithm presented here is capable of correctly handling the interplay between function and functions calls, side-effects, and short-circuit expression evaluation. The ability to deal with these features is required in reverse engineering of legacy systems, where code often contains side-effects.

The implementation of our slicing algorithm was achieved using a language called WSL [67]. WSL is both the language that the slicer was written as well as the object language to be sliced. The reason for our choice is that WSL has a built in WSL parser that can be called from within a WSL program as well as a whole transformation system which is useful for the simplification of slices. Transformations were not a major design criterion of most popular programming languages, they are difficult to transform. However, WSL (wide spectrum language) and transformation theory form the basis of the ’Maintainer’s
Assistant’s tool [83] used for analysing programs by transformations. WSL is also the basis of the FermaT transformation system [82]. The FermaT transformation system applies correctness-preserving transformations to programs written in WSL language. It is an industrial-strength engine with many applications in program comprehension and language migration, it has been used in migration IBM assembler to C and to COBOL [66]. Low-level programming constructs and high-level abstract specifications are both included in WSL language; hence the transformation of a program from abstraction specification to a detailed implementation can be expressed in a single language. The syntax and semantics of WSL are described in [67].

A denotational interprocedural slicing algorithm for programs in the presence of side-effects is the main contribution of this chapter. The slicer is given in Section 6.2. we first discuss slicing in the presence of side-effects.

### 6.1 Slicing in the Presence of side-effects

Slicing in the presence of side-effects is complicated because of the necessity to translate the slicing criterion into and out of function calls.

The slice of an assignment \( x = y \); with respect to a variable \( z \) is empty. The resulting slice of an assignment \( x = \text{f}(\cdot) \); with respect to a variable \( z \) is not always empty as there are two cases to consider: The first one is when the function \( \text{f} \) is side-effect-free. In this case, the resulting slice with respect to \( z \) is empty. However, if \( \text{f} \) contains a side-effect on \( z \), the assignment \( x = \text{f}(\cdot) \) is kept in the slice but we must still slice the body of the function \( \text{f} \) (keeping only the statements which affect the final value of the variable \( z \)).

In Figure 6.1, the right most fragment shows the resulting slice of the program \( P_1 \) with respect to \( z \). The assignment \( y = 1; \) in the body of the function \( \text{f} \) is deleted, because it does not contribute the the final value of the variable \( z \).
### 6.2 Interprocedural Slicing

The main denotational algorithm for interprocedural slicing of programs with side-effects is given in Section 6.2.

#### 6.2 Interprocedural Slicing

In many languages, function calls can occur either as expressions or as statements. Since such function calls may have side-effects, we need to be able to slice *expressions* as well as *statements*.

In this section a denotational interprocedural slicing algorithm is defined. The characteristics of the subroutines considered are based on those in WSL [67]. In WSL, functions may have both *value* and *var* parameters. Multiple *return* statements are not allowed. The value returned by a function is given by a single expression occurring at the end of the function’s body. Our system cannot at present handle recursive functions or multiple

<table>
<thead>
<tr>
<th>Original program $P_1$</th>
<th>Slice of $P_1$ w.r.t. $z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>int $x, y, z$;</td>
<td>int $x, z$;</td>
</tr>
<tr>
<td>int $f()$</td>
<td>int $f()$;</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>y = 1;</td>
<td>z = 1;</td>
</tr>
<tr>
<td>z = 1;</td>
<td>return $z$;</td>
</tr>
<tr>
<td>return $z$;</td>
<td></td>
</tr>
<tr>
<td>main()</td>
<td>main()</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>$x = f()$;</td>
<td>$x = f()$;</td>
</tr>
<tr>
<td>printf(&quot;%d&quot;, $z$);</td>
<td>printf(&quot;%d&quot;, $z$);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 6.1: Program $P_1$ and its corresponding slice w.r.t. $z$
First : N × [E] → E
First(i, L) = L[1], . . . , L[i−1]

E : [E] → E
E(L) = L[1], . . . , L[length(L)]

tempV : V × Names → V
tempV(x, f) = x_f

Temp_L : [V] × Names → [V]
Temp_L(L, f) =
  [tempV(L[1], f), . . . , tempV(L[length(L)], f)]

Temp_S : [V] × Names → P(V)
Temp_S(L, f) =
  {tempV(L[1], f), . . . , tempV(L(length(L)), f)}

Rename : [V] × [V] × S → S
Rename(L1, L2, S) = [L1 → L2]S
where [L1 → L2]S means for each i, substitute L1[i] by L2[i] in S

Figure 6.2: Auxiliary functions
returns.

We consider function calls to be atomic and therefore will remain in their entirety or be completely deleted. However, it is the bodies of the functions which can be simplified in the case where at least one of their corresponding calls remains in the slice.

Slicing across functions and functions-calls is complicated by side-effects. An expression $E$ can have side-effects upon the set of variables of interest. Therefore we need to work out, $\mathcal{N}(E, V)$, the set of variables whose initial values affect the final value of variables in $V$ when $E$ is executed and the set, $\mathcal{D}(E)$ of variables whose initial values determine the outcome of the value of the expression $E$.

As in the case of Hausler’s denotational approach [43], two functions are required:

$\mathcal{S}(P, V)$, the slice of program $P$ with respect to the set of variables $V$, which takes a statement $P$ and a set of variables $V$ and returns the resulting slice of $P$ with respect to $V$.

$\mathcal{N}(P, V)$, the needed set with respect to the set of variables, $V$, of program $P$. This is a function which returns the set of variables, whose initial values affect the final values of variables in $V$ when $P$ is executed.

### 6.2.1 Slicing Expressions

The algorithm for slicing expressions is given in Figure 6.3. It shows how to work out $\mathcal{N}$ and $\mathcal{D}$ for different types of expressions. We now consider each case in turn:
Slicing Compound Expressions

Since expressions can have side-effects, the order in which expressions are evaluated has to be considered. Therefore in order to determine $O(E)$ and $N(E, V)$, for compound expressions we need to know the order in which the sub-expressions of $E$ are evaluated. We assume sub-expressions are evaluated from left to right. Slicing compound separated expressions involves considering the three followings cases:

1. **Comma operator separated expressions**
   Comma expressions are always evaluated from left to right. $N(E, V)$ and $O(E)$ for comma expressions is given by formulae (5) and (6).

2. **Arithmetic operator separated expressions**
   Rules (7) and (8) in 6.3 show $N(E, V)$ and $O(E)$ for arithmetic operator separated expressions.

3. **Boolean operator separated expressions**
   For Boolean expressions, the issue of side-effects is complicated further by short circuit evaluation. In evaluating the boolean expression $B_1 \text{ op } B_2$, it is possible that only $B_1$ gets evaluated. In 6.3, rules (9) and (10) shows how to work out $N$ and $O$ of Boolean operator separated expressions.

Slicing Function Call Expressions

A function call expression is of the form $f(A_{val}, A_{var})$, where $A_{val}$ and $A_{var}$ are respectively the lists of actual value and var parameters. A function definition is of the form $f(F_{val}, F_{var}, S, e)$, where $F_{val}$, $F_{var}$, $S$ and $e$ are, respectively, the lists of formal value and var parameter, the body of the function and the return expression of the function. We only consider functions with one return statement.
\[ \mathbb{N} : \mathbb{E} \times \mathcal{P}(\mathbb{V}) \rightarrow \mathcal{P}(\mathbb{V}) \]
\[ \mathcal{D} : \mathbb{E} \rightarrow \mathcal{P}(\mathbb{V}) \]
\[ \alpha : \text{Names} \rightarrow \mathcal{P}(\mathbb{V}) \]
\[ \beta : \text{Names} \rightarrow \mathcal{P}(\mathbb{V}) \]

**SideEffect** : \( \mathbb{E} \times \mathcal{P}(\mathbb{V}) \rightarrow \{T, F\} \)

### Constants

1. \( \mathbb{N}(E, V) = V \)
2. \( \mathcal{D}(E) = \emptyset \)

### Variables

3. \( \mathbb{N}(E, V) = V \)
4. \( \mathcal{D}(E) = E \)

### Comma Expressions

5. \( \mathbb{N}(E_1, E_2, V) = \mathbb{N}(E_1, \mathbb{N}(E_2, V)) \)
6. \( \mathcal{D}(E_1, E_2) = \mathcal{D}(E_1) \cup \mathbb{N}(E_1, \mathcal{D}(E_2)) \)

### Arithmetic Expressions

7. \( \mathbb{N}(E_1 \text{ op } E_2, V) = \mathbb{N}((E_1, E_2), V) \)
8. \( \mathcal{D}(E_1 \text{ op } E_2) = \mathcal{D}((E_1, E_2), V) \)

### Boolean Expressions

9. \( \mathbb{N}(B_1 \text{ op } B_2, V) = \mathbb{N}(B_1, V) \cup \mathbb{N}(B_1, B_2, V) \)
10. \( \mathcal{D}(B_1 \text{ op } B_2) = \mathcal{D}(B_1, B_2) \).

### Function Calls

11. \( \mathbb{N}(f(A_{\text{val}}, A_{\text{var}}), V) = \mathbb{N}(A_{\text{val}}, (V' \setminus \text{Temp}_S(F_{\text{val}}, f)) \cup \bigcup_{i=1}^{n} (C_i) \)

\[
\begin{align*}
V' &= \mathbb{N}(S', \mathbb{N}(e, V)) \\
S' &= \text{Rename}(F_{\text{val}}, F_{\text{val}}, A_{\text{var}}, \text{Temp}_L(F_{\text{val}}, f), S) \\
\text{where} & \begin{cases} 
\text{and } e \text{ are the body and the return expression of } f \\
C_i = \begin{cases} 
\mathbb{N}((\text{First}(i, A_{\text{val}}), \mathcal{D}(A_{\text{val}}[i])) & \text{if } \text{Temp}_L(F_{\text{val}}, f)[i], V' \\
\emptyset & \text{otherwise}
\end{cases}
\end{cases}
\end{align*}
\]

11.1. \( \alpha(f) \rightarrow \alpha(f) \cup \text{Rename}(\text{temps}(F_{\text{val}}, f), F_{\text{val}}, S(S', V')) \)

12. \( \mathcal{D}(f(A_{\text{var}}, A_{\text{var}})) = \mathbb{N}(A_{\text{val}}, (V' \setminus \text{Temp}_S(F_{\text{val}}, f)) \cup \bigcup_{i=1}^{n} (C_i) \)

\[
\begin{align*}
V' &= \mathbb{N}(S', \mathcal{D}(e')) \\
e' &= \text{Rename}(F_{\text{var}}, F_{\text{val}}, A_{\text{var}}, \text{Temp}_L(F_{\text{val}}, f), e) \\
S', e \text{ and } C_i \text{ are defined above}
\end{align*}
\]

12.1. \( \alpha(f) \rightarrow \alpha(f) \cup \text{Rename}(\text{temps}(F_{\text{val}}, f), F_{\text{val}}, S(S', V')) \)

13. **SideEffect**\((f(A_{\text{val}}, A_{\text{var}}), V) = \begin{cases} 
T & \text{if } S(S', V') \neq \text{skip or } \exists i \text{ such that } \text{SideEffect}(A_{\text{val}}[i], V) \\
F & \text{otherwise}
\end{cases} \)

where

\( S' \text{ and } V' \) are defined in rule 11.

Figure 6.3: Interprocedural Slicing of Expressions
Slicing a function call, given in Figure 6.3, involves computing N and D for the function call and slicing the corresponding function definition.

\[ N(E, V) \text{ for expressions of form } f(A_{val}, A_{var}) \text{ (a call the a function } f) \text{ is given by formula (11) in Figure 6.3.} \]

Let \( S \) and \( e \) be the body and the return expression of the function \( f \): First we tag every formal value-parameter occurring in \( S \) and then replace every formal var-parameter occurring in \( S \) by its corresponding actual one. As a result of doing this, we get a new statement \( S' \). We then slice \( S' \) with respect to \( N(e, V) \) to get \( V' \). In order to proceed, we remember that evaluating the actual value-parameters of a function call can contain a side-effect. Each element of \( V' \) which is not a tagged value parameter can be affected by whole the list \( A_{val} \). Every tagged element in \( V' \) can only be affected by the actuals evaluated before it. This is captured by the \( C_i \) in Figure 6.3.

Rule (12) in Figure 6.3 computes \( D(E) \) for function calls. It is almost identical to computing \( N(E, V) \). The same tagging occurs. The only difference is that, in this case, \( V' = N(S', D(e')) \) where \( e' \) is just the return expression of \( f \) with every formal var-parameter occurring in \( e \) replaced by its corresponding actual one.

Each time a function call \( f(A_{val}, A_{var}) \) is encountered, the body of its corresponding function definition has to be sliced with respect to a set of variables \( V \), taking into account the relationship between formal parameters and actual parameters. To be able to slice the function definition \( f \), a record, \( \alpha(f) \), is kept of all statements in the body of the function \( f \), needed at each of the corresponding calls. During slicing each time we come across a call to a function \( f \), \( \alpha(f) \) is updated to reflect that parts of \( f \)’s body that are needed in the slice. Rules (11.1) and (12.1) in Figure 6.3 show how \( \alpha(f) \) is updated each time a function call is sliced.
6.2.2 Interprocedural Slicing of Statements

Before we can discuss the interprocedural slicing of statements, we need to define how to ascertain whether an expression \( E \) has a side-effect on a set of variables \( V \) (See Rule 13 in Figure 6.3). If \( E \) does not contain a function call it cannot have a side-effect on \( V \). The only way that \( E \) can side-effect an element of \( V \) is if at least one of the function calls in \( E \) has a side-effect on an element of \( V \). A function call has a side-effect on an element of \( V \) if and only if when we tag the body (as described in Section 6.2.1) and slice it with respect to \( V \) we do not get skip. The slice(\( S \)) and needed set(\( N \)) with respect to set of variables \( V \) of each form of statement is given in Figure 6.4.

Assignment Statements

Rules (1) and (2) in Figure 6.4 compute \( N(x = e, V) \) and \( S(x=e,V) \), we have three cases to consider

1. If \( x \) is not an element of \( V \) and \( e \) has no side-effect on \( V \), then \( N(x = e, V) \) is just \( V \). The assignment, in this case is deleted.

2. If \( x \) is not an element of \( V \) and \( e \) may have a side-effect on \( V \), then \( N(x = e, V) = N(e, V) \). In this case the assignment is kept in the slice.

3. If \( x \) is an element of \( V \), then to compute \( N(x = e, V) \) we remove \( x \) from \( V \) and add in all the variables affecting the final value of \( e \). In this case, the assignment is kept in the slice.

Conditionals

Rule (3) and (4) in Figure 6.4 compute the needed set and the slice for an if statement. First we slice both the true and the false parts with respect to \( V \). Again there are three
### 6.2 Interprocedural Slicing

<table>
<thead>
<tr>
<th>Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) ( N(x = e, V) = \begin{cases} V &amp; \text{if } x \notin V \land \neg \text{SideEffect}(e, V) \ N(e, V) &amp; \text{if } x \notin V \land \text{SideEffect}(e, V) \ N(e, V \setminus {x}) \cup \mathcal{D}(e) &amp; \text{if } x \in V \wedge x \in V \end{cases} )</td>
</tr>
<tr>
<td>(2) ( S(x = e, V) = \begin{cases} \text{skip} &amp; \text{if } x \notin V \land \neg \text{SideEffect}(e, V) \ x = e &amp; \text{otherwise} \end{cases} )</td>
</tr>
</tbody>
</table>

### Conditionals

(4) \( N(\text{if } (B) S_1 \text{ else } S_2, V) = \begin{cases} V & \text{if } S(S_{i_{(1,2)}}, V) = \text{skip} \land \neg \text{SideEffect}(B, V) \\ N(B, V) & \text{if } S(S_{i_{(1,2)}}, V) = \text{skip} \land \text{SideEffect}(B, V) \\ N(B, N(S_1, V) \cup N(S_2, V)) \cup \mathcal{D}(B) & \text{if } S(S_{i_{(1,2)}}, V) \neq \text{skip} \end{cases} \)

(5) \( S(\text{if } (B) S_1 \text{ else } S_2, V) = \begin{cases} \text{skip} & \text{if } S(S_{i_{(1,2)}}, V) = \text{skip} \land \neg \text{SideEffect}(B, V) \\ \text{if } (B) S(S_1, V) \text{ else } S(S_2, V) & \text{otherwise} \end{cases} \)

### Whiles

(5) \( N(\text{while } (B) S, V) = \begin{cases} V & \text{if } \neg \text{SideEffect}(B, V) \land S(S, V) = \text{skip} \\ \text{fix } \lambda y \cdot N(\text{if } (B) S, y \cup (N(B, V) \cup \mathcal{D}(B))) & \text{otherwise} \end{cases} \)

(6) \( S(\text{while } (B) S, V) = \begin{cases} \text{skip} & \text{if } \neg \text{SideEffect}(B, V) \land S(S, V) = \text{skip} \\ \text{while } (B) S(S, N(\text{while } (B) S, V)) & \text{otherwise} \end{cases} \)

### Expression Statements

(7) \( N(e, V) \) (See 6.3)

(8) \( S(e, V) = \begin{cases} \text{skip} & \text{if } \neg \text{SideEffect}(E, e) \\ e & \text{otherwise} \end{cases} \)

### Function Definitions

(11) \( S(f(A_{\text{val}}, A_{\text{var}}, S, e), V) = \begin{cases} \text{skip} & \text{if there is no call to the function } f \text{ the slice} \\ f(F_{\text{var}}, f_{\text{val}}, \alpha(f), e) & \text{otherwise} \end{cases} \)

where \( \alpha(f) \) is defined in Figure 6.3.

Figure 6.4: Interprocedural Slicing of Statements
cases to consider:

1. If the slices of both subcomponents are empty and $B$ as no side-effect on $V$, then the whole if statement is deleted and the needed set of variables is just $V$.

2. If the slices of both subcomponents are empty and $B$ may have a side-effect on $V$, then the needed set of variables is $\mathbb{N}(B, V)$ and the resulting slice is just if $(B)$ In this case the needed set of variables is $\mathbb{N}(B, V)$.

3. If either of the slices of the subcomponents are not empty, then the resulting slice is if $(B)$ $S(S_1, V)$ else $S(S_2, V)$, and the needed set of variables is $\mathbb{N}(S_1, V) \cup \mathbb{N}(S_2, V) \cup \mathbb{O}(B)$

**Loops**

Interprocedural slicing of loops is similar to the intraprocedural case. The while loop can be deleted in the case where neither the predicate nor the body affect the set of variables $V$. In the case either the predicate or the body affects $V$ we must consider the affect of the predicate on the set of variables, $V$ as well as the variables that determine the value of the predicate (rules 5 and 6).

**Slicing Local Scope**

We write $\text{Loc}(W, S)$ where $W$ is a set of local variable names and $S$ is a statement to represent the effect of 'localising' the variables in $W$ for the scope of $S$. To slice $\text{Loc}(W, S)$ with respect to $V$, first all the local variables in $S$ are renamed with some new identifiers to give $S'$. The statement $S'$ is then sliced with respect to $V$. For the needed set we simply remove all local variables from the set $\mathbb{N}(S', V)$, and for the slice we simply 'undo' the renaming of the slice of $S'$ with respect to $V$. 
6.3 Related Work

Slicing Expression Statements

To compute the needed set of an expression statement we just use the rules for calculating $N$ for expressions given in Figure 6.3.

To compute $S$ for expression statements, we simply keep the expression statement in the slice if and only if it may have a side-effect on $V$. This function is defined in Figure 6.4.

Slicing Function Definitions

To slice a function definition we simply take the union of the slices of the function definition corresponding to each call to it (Rule 11 in Figure 6.4). If there is no call then the function definition will not be included.

6.3 Related Work

Hausler [43] presents a denotational program slicer for a very simple programming language without procedures. This alogrithm was described in Chapter 3. In Chapter 5 Hausler’s algorithm was proved to be correct.

Horwitz et al. [77] have introduced an interprocedural slicing algorithm as graph reachability over the system dependence graph (SDG). The SDG approach [77] is explained in details in Chapter 2. CodeSurfer [34] is a commercial interprocedural slicing tool based on reachability problem over SDG [77]. In order to slice a program $P$ using with respect to a program point $p$ and a variable $x$, CodeSurfer highlights all codes of the program $P$ that affect the value of the variable of interest $x$ at the program point $p$. It does not return an executable program. Furthermore, The cost of time to construct an SDG for a program is polynomial in the size of the program. In some application of slicing, for example, amor-
phous slicing\textsuperscript{38} a program needs to be sliced several times, using CodeSurfer we would need to construct a new \textit{SDG} after each transformation, this is very time consuming.

### 6.3.1 Advantages of Our Approach

The advantage of the denotational approach is that slicing can be expressed as mathematical transformations on abstract syntax without the need to introduce intermediate structures such as control flow graphs. Such definitions are highly amenable both to correctness proof and implementation in the functional style. Our algorithm not only returns an executable program, but also produces variable dependence information which is very useful in comprehension and testing \textsuperscript{40}. 

Chapter 7
Conclusion and Future Work

7.1 Conclusion

Program slicing [85] produces simpler programs from complicated ones and so can be thought of as form of program transformation. One reason that correctness proofs of traditional slicing algorithms are so difficult is that a semantics of the intermediate structure is required in addition to bi-directional mappings between programs and these intermediate structures.

The main aim of this thesis is to investigate slicing directly without any intermediate structures. We regard the intermediate structures as mere ‘implementation details’. Everything, the slicing algorithm and the semantics of the program language and the ‘correctness criteria’ of slicing are now expressed denotationally. This allows the possibility of using the full power and elegance of denotational semantics in definitions and correctness proofs.

In 1989, Hausler [43] introduced the idea of expressing a slicing algorithm denotationally. He defined an end-slicing algorithm using a number of denotational rules for a simple imperative language consisting of assignments, statement sequences, conditionals and loops but no subroutines. He did not prove his algorithm correct and no correctness proof can be found in the literature. This, naturally, became our initial goal.

In order to prove the correctness of Hausler’s Algorithm [43], a satisfactory definition of
correctness had to be given. According to Weiser [88], a program and its (end) slice must agree with respect to the set of variables in the slicing criterion. In other words, if we run the original program and the slice, then, in all states where the original terminates, the slice must also terminate with the same final values for the variables in the slicing criterion. This is the correctness criterion that needs to be proved for any slicing algorithm. The behaviour of the slice in states where the original does not terminate is left undefined. In fact, traditional slicing algorithms sometimes introduce termination: the standard semantics of a program is thus, less defined than the semantics of some of its slices. Because of this it is inappropriate to try to prove correctness of slicing properties using standard semantics [33]. This was observed and first discussed by Cartwright and Felleisen [15].

Central to slicing is the concept of variable dependence: the set of variables needed by a set of variables $V$ in program $P$. Intuitively, this is the set of variables whose initial value ‘may affect’ the final value of at least one variable $v$ in $V$ after executing $P$. One of our aims was to make the phrase ‘may affect’ semantically precise.

The semantic definition of variable dependence, clearly had to be consistent with Hausler’s slicing algorithm. That is, all variables that are ‘semantically needed’ must also be ‘Hausler Needed’. Standard semantics loses precision in the presence of infinite loops. Because of this, it is hard, if it is not impossible, to define neededness in terms of standard semantics. Much of our research involved trying to find a semantics which was both consistent with standard semantics and allowed us to define variable dependence satisfactorily.

This leads to a new lazy semantics defined in Section 4.1, Chapter 4 which is at the heart of our work. A semantics definition of variable dependence is relatively straightforward to do so in terms of our lazy semantics. Our semantics is, unsurprisingly, closely related to the semantics of Cartwright and Felleisen [15], as that was a semantics for program dependence graphs. It also bears some similarity to the transfinite semantics of Giacobazzi and Mastroeni [33]. Unlike semantics of Cartwright and Felleisen [15] and that of by Giacobazzi and Mastroeni [33], our lazy semantics is substitutive. Such a property is useful
to prove correctness for the kind of program transformations such as slicing [38, 9, 42].

As a demonstration of the applicability of our lazy semantics, in Chapter 4, Hausler’s Denotational Slicing Algorithm is proved correct with respect to the lazy semantic definition of a slice given in Section 4.1. Since our lazy definition of a slice is stronger than the standard one, this proves that Hausler’s Algorithm [43] is correct with respect to the standard definition too.

### 7.2 Future work

A future direction of our research is to attempt to extend the ideas presented in this thesis to programs with procedures. To do this, we must:

- Extend the lazy semantics to handle programs with procedures.
- Extend Hausler’s slicing algorithm to handle programs with procedures. That is, to define an *interprocedural* slicing algorithm denotationally and prove its correctness with respect to the lazy semantics introduced in this thesis.

### 7.2.1 Extending the Lazy semantics to Programs with Procedures

There are many issues involved in extending our semantics to handle procedures, including how to deal with local variables, parameters and, of course recursion. We assume for the moment that procedure definitions have no parameters or local variables. Without recursion the semantics can be defined as the semantics obtained by replacing all procedure calls by their bodies.

There are motivating ideas that help guide us toward a lazy semantics of procedure calls. They are:
1. The lazy semantics must be preserved by current slicing algorithms.

2. If a recursion can be replaced by loops, then the semantics must agree and also, as in
the non-procedural case, the lazy semantics must agree with the standard semantics
for terminating programs.

Let us first consider the issue of recursion.

**Slicing Recursive Procedures**

```
int x, y;
f()
{
    y = y + 1;
    f();
}
main()
{
    f();
    x = 0;
}
```

```
int x;
main()
{
    x = 0;
    f();
}
```

```
int y;
f()
{
    y = y + 1;
    f();
}
main()
{
    f();
}
```

A simple program \( P \). _slice of \( P \) w.r.t. \( x \).  Slice of \( P \) w.r.t. \( y \).

Figure 7.1: A simple C program \( P \) and its corresponding slices w.r.t. \( x \) and \( y \).

Slicing algorithms \([49, 50, 74]\) will delete any procedure for which the body does not affect
the variables of interest. For example, slicing the program \( P \) in Figure 7.1, with respect to
\( x \) will remove the procedure call \( f() \) as it has no effect on the final value of \( x \). However,
slicing the program \( P \) with respect to \( y \) will keep the call \( f() \) in the slice. For the program
\( P \) and its slice with respect to \( x \) to have the same semantics we require the semantics of
\( f() \) with respect to \( x \) to be the same as the semantics of \texttt{skip} \texttt{.} Clearly, the tail-recursive
procedure \( f() \) in the program \( P \) can be replaced by a while loop. The program \( P' \) in
Figure 7.2 shows a program that is equivalent to $P$ the procedure call is replaced by an equivalent while loop.

\begin{verbatim}
int x, y;
main()
{
    while (y>0)
    {
        y:=y+1;
    }
    x:=0 ;
}
\end{verbatim}

A program $P'$.

\begin{verbatim}
int x;
main()
{
    x:=0 ;
}
\end{verbatim}

Slice of $P'$ w.r.t. $x$.

\begin{verbatim}
int y;
main()
{
    while (y>0)
    {
        y:=y+1;
    }
}
\end{verbatim}

Slice of $P'$ w.r.t. $y$.

Figure 7.2: A program $P'$ equivalent to $P$ in Figure 7.1 and its corresponding slices w.r.t. $x$ and $y$.

The semantic equivalence of some recursive procedure with while loops becomes the prime motivation of our lazy semantics for procedure calls. In order to handle recursive procedures, it may be possible to define the semantics of procedure calls defined, as in the case of loops, as the \emph{lim inf} of all the corresponding procedure unfoldings. Given a state $\sigma$ and a variable $x$ the final lazy value of $x$ after executing a call to procedure $f$ starting in state $\sigma$ is the limit of all the values of $x$ after executing each of the $f$'s unfoldings. If the limit does not exist, then we define the final lazy value to be $\perp$. Here, as in the case of loops, we mean the limit with respect to a discrete metric i.e. for the limit to exist, there must exist an $N \in \mathbb{N}$ such that all unfoldings greater than $N$ give the same value for $x$ in $\sigma$. If this is the case we say the value of $x$ stabilises after $N$ unfoldings.

In order to handle procedures, we extend our language by now thinking of a program as a sequence of procedure definitions. A procedure definition $P$ can be written as $P : S$ where the name $P$ is its name and $S$ is its body. The body is just a sequence of all the constructs of our base language plus procedure calls. Procedures are allowed to call
themselves directly or indirectly. The program language under consideration is illustrated in Figure 7.3.

\[
\begin{align*}
Program : & \text{ProcDef} \parallel \text{ProcDef; Program} \\
\text{ProcDef} : & \text{ProcName} : \text{Statement} \\
\text{ProcName} : & \text{String} \\
\text{ProcCall} : & \text{String Body} : \text{Statement} \\
\text{Statement} : & \text{skip} \parallel \text{abort} \parallel \text{Assignment} \parallel \text{Conditional} \parallel \text{Loops} \parallel \\
& \text{ProcCall} \parallel \text{Statement; Statement}
\end{align*}
\]

Figure 7.3: The program language under consideration.

Unfoldings of procedures

In order to deal with recursion, we could try to unfold procedures in a similar way to how we unfolded while loops in the previous chapter.

**Definition 32 (Unfoldings of procedures):**

Let \( P_i \) be all the procedure definitions and \( S_i \) their corresponding bodies. The unfolding of the procedures \( P_i \) is defined as follows:

\[
\begin{align*}
\mathcal{W}_0(P_i) &= \text{abort} \\
\mathcal{W}_{n+1}(P_i) &= S_i[\text{call}P_j \leftarrow \mathcal{W}_n(P_j)]
\end{align*}
\]

where \( \mathcal{W}_{n+1}(P_i) = S_i[\text{call}P_j \leftarrow \mathcal{W}_n(P_j)] \) is just the body of \( P_i \), \( S_i \), where all procedure calls are replaced by the \( n \)th unfolding of their corresponding procedure definitions.

In other words, to produce the \((n + 1)\)th unfolding of procedure \( p \), we replace all calls in \( p \)
by their corresponding $n^{th}$ unfoldings.

```c
int x, y;
g() {
    x = y;
}
p() {
    y = z;
g();
}
```

Figure 7.4: A simple program $P$.

<table>
<thead>
<tr>
<th>$W_0(g)$</th>
<th>$W_1(g)$</th>
<th>$\ldots$</th>
<th>$W_n(g)$, $n \geq 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>x := y</td>
<td>x := y</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.5: Unfoldings of the procedure $g$ in Figure 7.4.

Figure 7.5 shows the unfoldings of the procedure $g$ of the program in Figure 7.4. The zeroth unfolding is always abort statement, which the same as in the case of while loop. As the body of the procedure $g$ does not contain any procedure call, the $n^{th}$ unfolding for all $n \geq 1$ is just the body of the procedure $g$.

```c
<table>
<thead>
<tr>
<th>$W_0(p)$</th>
<th>$W_1(p)$</th>
<th>$W_2(p)$</th>
<th>$\ldots$</th>
<th>$W_n(P)$, $n \geq 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>y := z</td>
<td>y := z</td>
<td>y := z</td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td>x := y</td>
<td>x := y</td>
<td>x := y</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 7.6: Unfoldings of the procedure $p$ in Figure 7.4.

Figure 7.6 shows the unfoldings of the procedure $p$ of the program in Figure 7.4. Again, the zeroth unfolding is always abort, the first unfolding is just the body of $p$ where the
call to $g$ is replaced by `abort`. For all $n \geq 2$ the $n^{th}$ unfolding of the procedure $p$ is just
$y := z; x := y$.

We now define the standard and lazy semantic of procedure calls in terms of the unfoldings of their corresponding procedure definitions.

**Definition 33 (standard meaning):**

The standard meaning of a call to procedure $P$ is defined as follows:

$$\mathcal{M}[\text{call } P] = \lambda \sigma \cdot \bigcap_{i=0}^{\infty} \mathcal{M}[\mathcal{W}_i(P)] \sigma.$$ 

**Definition 34 (Lazy meaning):**

The lazy meaning of a call to procedure $P$ could be defined as follows:

$$\mathcal{M}_L[\text{call } P] = \lambda \sigma \cdot \bigcap_{i=0}^{\infty} g_i \sigma \quad \text{where} \quad g_i \sigma = \bigcap_{n=i}^{\infty} \mathcal{M}_L[\mathcal{W}_n(P)] \sigma.$$ 

From the definition of $g_i$ we can see that $g_i$ is an increasing chain, $g_i \subseteq g_{i+1}$, then its least upper bound exists.

One of the main properties any semantics has to satisfy if it can be useful to prove correctness of slicing algorithm is that it should agree with the standard semantics in states where the program terminates. The following theorem shows that the lazy semantics given in Definition 34 satisfies this property.

**Theorem 7.2.1:** Let $P$ be a program and $\sigma$ be a state in $\Sigma$, then,

$$\mathcal{M}[\text{call } P] \sigma \neq \bot \implies \mathcal{M}_L[\text{call } P] \sigma = \mathcal{M}[\text{call } P] \sigma.$$
Proof. Let \( \sigma \) be a state in \( \Sigma \) with \( M[\text{call } P] \sigma \neq \bot \). By Definition 33 there exists an integer \( n \) such that for all \( m \geq n \) we have \( M[\text{call } P] \sigma = M[\mathcal{W}_m(P)] \sigma \neq \bot \). For all \( i \geq 0 \) the \( i^{th} \) unfolding of the procedure \( P \), \( \mathcal{W}_i(P) \), is made only of statements of our base language, abort, skip, assignment, conditionals, sequences and loops. Hence, By Theorem 4.1.1 For all \( m \geq n \) we have

\[
M[\mathcal{W}_m(P)] \sigma = M_L[\mathcal{W}_m(P)] \sigma.
\]

And by Definition 34 we have

\[
M_L[\text{call } P] \sigma = \bigcup_{i=0}^{\infty} \bigcap_{j=i}^{\infty} M_L[\mathcal{W}_j(P)] \sigma = \bigcup_{i=n}^{\infty} \bigcap_{j=i}^{\infty} M_L[\mathcal{W}_j(P)] \sigma.
\]

Hence,

\[
M_L[\text{call } P] \sigma = \bigcap_{j=n}^{\infty} M[\mathcal{W}_j(P)] \sigma = M[\mathcal{W}_m(P)] \sigma, \quad \forall m \geq n.
\]

Thus completing the proof. \( \square \)

We also require the lazy meaning of a tail recursive function to be the same as the lazy meaning of the corresponding loop. i.e.

**Lemma 7.2.2:** Let \( S \) be a program and \( P \) be a recursive procedure with if \((B)\) then \( S;\) call \( P \), its corresponding body. Then, \( M_L[\text{call } P] = M_L[\text{while}(B) \ S] \).

Unfortunately, the semantics given in definition 34 breaks down here as can be seen by the following example: If Lemma 7.2.2 is true then the programs in Figures 7.7 and 7.8 should have the same lazy semantics. Unfortunately, the loop version maps \( x \) to \( \bot \) whereas the the recursive version maps \( x \) to \( 0 \). This is because in order to work out the meaning of a nested loop, the inner loop is fully unfolded before the outer loop is unfolded at all. In the case of recursive functions as defined in Definition 34 the two recursive functions are unfolded ‘together’. After each such unfolding it can be seen the final value of \( x \) is zero. So in the limit the final value of \( x \) will still be zero, not \( \bot \) as is the case for loops.
7.2 Future work

$$\begin{align*}
    \text{int } x; \\
x := 0; \\
\text{while(True)} \\
\quad \{ \\
\quad \quad \text{while(true)} \\
\quad \quad \quad \{ \\
\quad \quad \quad \quad x := 1 - x; \\
\quad \quad \quad \} \\
\quad \quad \text{if (x=1) then } x := 0; \\
\quad \}
\end{align*}$$

Figure 7.7: A program $P$ with two nested loops.

7.2.2 Extending Hausler's slicing algorithm to Programs with Procedures

Most slicing algorithms are defined using different intermediate program graph representations.

For multi-procedure programs, Horwitz et al [50] have defined the system dependence graph, which consists of a program dependence graph for each procedure of the program. They introduce several nodes to model procedure calls and parameter passing. Parameters are passed by value-result and accesses to global variables are modelled via additional parameters of the procedure. They then introduced an interprocedural slicing algorithm as graph reachability problem over the system dependence graph. Their algorithm is discussed in detail in Section 2.5.

Ouarbya et al [74] have extended Hausler’s algorithm to handle programs with non-recursive procedures. Future work will attempt to extend this to programs with recursive procedures and prove correctness for this case. Clearly this cannot be achieved until the
int x; g()
{
  if (True) then  x:=1-x;
  g();
}
f()
{
  if (True) then  g();
  if (x=1) then  x:=0;
  f();
}
main()
{
  x:=0 ;
  f();
}

Figure 7.8: An $P$-equivalent, $P'$, with two recursive functions.

problems of extending the lazy semantics to procedures have been solved.
Bibliography


[34] Grammatech Inc. The codesurfer slicing system, 2002.


