Dataflow Minimal Slicing

by

Sebastian Danicic

BSc. Pure Mathematics (London University)

MSc. Computation (Oxford University)

A thesis submitted in partial fulfillment of the

requirements of the University of North London

for the degree of

Doctor of Philosophy

May 20, 1999

To Aurora

Abstract

A *slice* of a program p with repsect to a variable x is a set, S, of statements from p such that every statement that affects x is in S. S may, but need not, contain statements that do not affect x.

Slicing enables large programs to be decomposed into ones which are smaller and hence pontentially easier to analyse, maintain and test. It is desirable that slicing algorithms produce slices that are 'as small as possible'. A *statement minimal* slice, S, with respect to p and x is a set which contains *only* statements that affect x. Statement minimal slices are known not to be computable.

Slicing algorithms traditionally work at the *dataflow level*, i.e. the only information used about each expression in the program being sliced is the set of variables referenced by the expression. As a result, such algorithms cannot distinguish between programs with identical structure up to expressions, where corresponding expressions reference the same set of variables. Such algorithms, are thus, in effect, not working on single programs but on sets of *dataflow equivalent* programs.

The slices produced by these algorithms are not, however, *dataflow minimal.* This means that for some programs, p, the slice produced by such an algorithm with respect to x contains statements that do not affect x in *all* programs in the *dataflow equivalence class* of p.

This thesis is an investigation into the question: Are dataflow minimal slices computable?

We introduce a definition of a form of dataflow minimal slice and develop an algorithm for computing it which we prove to be correct for loop free programs.

For programs, p, containing loops we prove that there exists an integer n, which we call the *maximal unfolding number* for p, where the dataflow minimal slice of p is the same as the dataflow minimal slice of its *mth unfolding* for all $m \geq n$. An unfolding is, by definition, loop free and therefore its dataflow minmal slice can be computed. The problem of computing a dataflow minimal slice of p is thus reduced to the problem of finding the maximal unfolding number of p. We implement a slicing algorithm based on unfolding which repeatedly unfolds p until there are no further additions to p's slice set. This algorithm is guaranteed to produce dataflow minimal slices provided that reaching this *stable state* implies that p's maximal unfolding number has been reached.

Acknowledegements

First and foremost, I would like to thank my main supervisor Dr Mark Harman of Goldsmiths' College, London University.

He has been the perfect supervisor both technically and practically. I was inspired by his broad knowledge and deep understanding of the subject matter and I benefited greatly from his skills in managing research. His tenacity and encouragement were unremitting throughout.

Among my most treasured experiences are the many hours each week that we spend discussing interesting issues, not only related to the subject matter of this thesis. Long may they continue!

I would like to thank my other supervisors, Professor Malcolm Munro of the University of Durham and Professor Dan Simpson of Brighton University for all their help and encouragement.

I would also like to thank the following people:

Professor Yau Jim Yip and Professor Ian Haines of the University of North London for their support and encouragement and for providing me with the conditions necessary to complete this thesis.

Dr Ross Paterson of City University, for introducing me to the delights of Debian Linux, for his beautiful Hope interpreter and for never failing to know the answer to every question I have asked.

Tom Reps of the University of Wisconsin and Tim Teitelbaum of Cornell University for suggesting that I use schemas.

Mrs. Yoga Sivagurunathan of the University of North London for producing the lovely diagrams in the Parallel Algorithm example (Section 2.3.4, page 44).

Dr John Howroyd of Goldsmiths' College, London University, for the 'John Howroyd' example (Section 3.13.1, page 113).

My 'room mates', Paul Fairney and Ryan Newdick who have both been exceedingly encouraging and supportive and have had to put up with months of dirty coffee cups without complaining!

Lastly, I would like to thank Aurora and Ivan both for proof reading and for their many helpful comments and suggestions.

Contents

List of Figures

Part I

Dataflow Minimal Slicing

Chapter 1

Introduction

1.1 Program Slicing

The underlying feature of all forms of slicing is that from a big complicated program, a smaller simpler slice is obtained. Analysis of some of the properties of the big program is thereby translated into the potentially easier problem of analysing the slice.

Program Slicing was introduced by Mark Weiser in his PhD thesis [92]. Informally, a program p is sliced with respect to a **slicing criterion** which is a pair (V, i), where V is a set of variables and i is a 'point[1]' in the program. The slice s of p is obtained from p by deleting statements and has the property that p and s 'behave the same' with respect to the slicing criterion (V, i).

1.1.1 An Example of Slicing

Consider program $p_{1.1}$ in Figure 1.1(page 22). Slicing $p_{1.1}$ with respect to the set of variables $\{\mathtt{x}\}$ at the end of the program would yield the program $p'_{1.1}$. *Syntactically* the slice, $p'_{1.1}$, has been obtained from the original, $p_{1.1}$, by deleting statements. A *semantic relationship* exists between $p_{1.1}$ and $p'_{1.1}$ in the sense that in all initial states they both result in the same final value of the variable `x`.

[1]We imagine all the statements of the program to be labelled.

```
begin
 x:=y;
 if x=3
 then
    begin
    c:=y;
    x:=25
    end;
 i:=i+1
end
```

```
begin
 x:=y;
 if x=3
 then


    x:=25


end
```

Figure 1.1: Program $p_{1.1}$ and its slice, $p'_{1.1}$

1.2 Applications of Slicing

Slicing has many applications including

- Program Comprehension [10, 31, 50, 51]
- Program Maintenance [11, 18, 20, 26, 38, 41, 40, 39, 48, 79, 85, 95]
- Program Debugging [94, 4, 84, 66, 77, 86]
- Testing [9, 13, 45, 46, 66]
- Re–engineering [75, 85] and Component Re–use [8]
- Program Integration [57]
- Software Metrics [12, 81, 79, 80, 74, 49, 76]

Tip [89] and Binkley and Gallagher [15] provide detailed surveys of the paradigms, applications and algorithms for program slicing.

A major aim of slicing is to delete as many statements from a program as possible in producing its slice as, certainly for program comprehension, maintenance, debugging and testing there is little doubt that, in general, everything else being equal, small programs are easier to understand and maintain than larger ones. Much of the literature on program slicing is concerned with improving the algorithms for slicing both in terms of size of slice (the smaller the better) and efficiency of the slicing algorithm.

Although the ultimate goal of statement minimal slicing is known not to be computable [92], much work [47, 35, 87, 20, 26] has been done to produce more precise dependence information and more accurate slices than those produced by Weiser's slicing algorithm [92].

1.3 Dataflow analysis

Weiser's algorithm works on control flow graphs [53] rather than programs. The control flow graph of a program is a structure where each 'statement' in the program is represented by a node. See Figure 1.3(page 25) for an example of a control flow graph. There is an arc connecting node n to node m if and only if 'execution can pass' from the statement corresponding to node n to the statement corresponding to node m. Each node of the control flow graph is annotated with two sets: the set of variables *defined* and the set of variables *referenced* by the corresponding statement. No other information about the program is recorded in the control flow graph[2].

Dataflow analysis [92, 53], by definition, is the act of inferring properties about a program from its control flow graph alone. Dataflow analysis is, thus, fairly limited. We cannot, for example, tell by looking at a program's control flow graph when two expressions in the program are equal, nor can we use any form of expression simplification. All the information required to do such things has been 'abstracted away' in converting the program into a control flow graph. All the approaches [47, 35, 87, 20, 26] cited above lie outside the realm of dataflow analysis; the improvements in precision that they exhibit result from the fact that they use *more* information about each expression in a program than simply the *referenced variables*. Weiser's algorithm, on the other hand, *is* an example of dataflow analysis.

1.4 The Problem

Weiser [92] noticed that his algorithm is not *dataflow minimal*. It included what appeared to be unnecessary statements in slices. Importantly, the fact that these nodes were unnecessary could be observed using dataflow analysis alone.

In this thesis, we set out to answer the following questions:

[2] This definition of a control flow graph is used throughout this thesis.

1. Why do dataflow slicing algorithms, like Weiser's produce slices that are not dataflow minimal?

2. Do algorithms for producing dataflow minimal slices exist?

1.5 Examples of the Dataflow Minimality Problem

An example illustrating the problem (Weiser's was more complicated) is given in Figure 1.2(page 24).

```
1   while i<0
    do
    begin
2      if c=3
       then
       begin
3         c:=4;
4         x:=5
       end;
5      i:=i+1
    end
```

Figure 1.2: Program $p_{1.2}$

Using Weiser's definition [93], an end–slice of $p_{1.2}$ with respect to the variable `x` is any program p obtained from $p_{1.2}$ by statement deletion which terminates whenever $p_{1.2}$ does, with the same final value for `x`.

The control flow graph of $p_{1.2}$ is given in Figure 1.3(page 25).

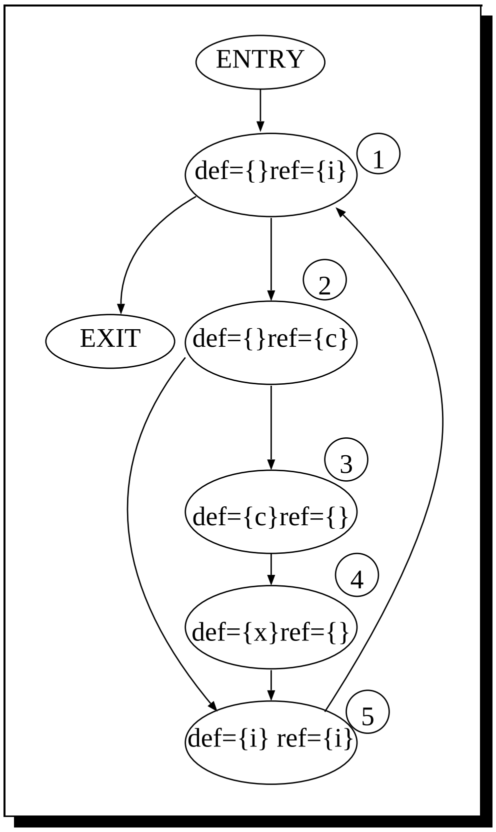


Figure 1.3: The Control Flow Graph of $p_{1.2}$

Using Weiser's algorithm, slicing on `x` at the end of the program, gives the whole control flow graph which, by definition, is a legal slice. It turns out that the smaller program whose control flow graph is given in Figure 1.4(page 26) is also a slice with respect to `x` at the end of the program i.e. every program whose control flow graph is the one in Figure 1.3(page 25) will 'behave the same' with respect to the final value of `x`, as the corresponding program whose control flow graph is the one in Figure 1.4(page 26).

Consider the control flow graph in Figure 1.3(page 25). The constant assignment at node 3 is executed if and only if the constant assignment at node 4 is executed. Having been assigned a constant value, the value of `x` cannot be further changed by the body of the loop. The initial value of `c` is important, but not the later assignment to it. There is no execution path from the entry node of the control flow graph to its exit node where the constant assignment at node 3 has an effect on the final value of `x`. Node 3, therefore, cannot affect the final value of

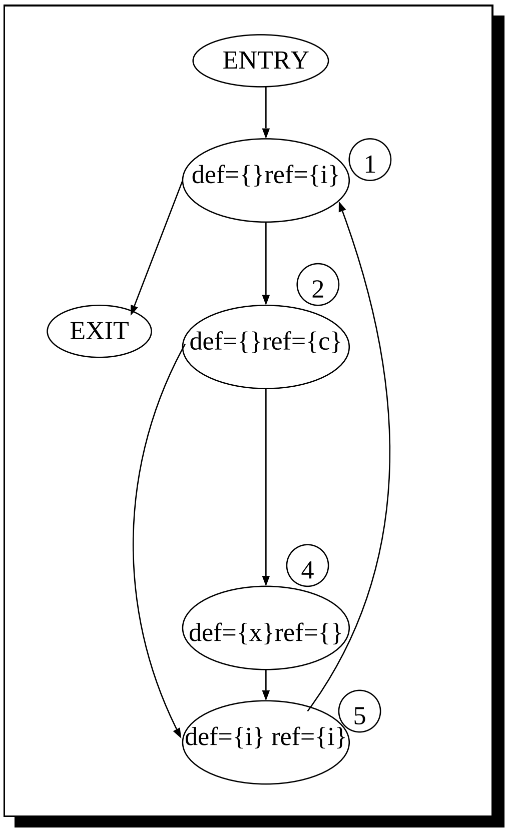


Figure 1.4: Dataflow Minimal Slice of $p_{1.2}$

x and thus need not be included in the slice.

Importantly, this argument depends not on the original program being sliced, but only on its control flow graph. We have not used any information about the expressions in the program apart from the set of variables they reference. This smaller slice is, in fact, dataflow minimal. We cannot remove any more nodes without affecting the final value of `x`. The comparison between the slice produced by Weiser's algorithm and the dataflow minimal one is given in Figure 1.5(page 27).

Another interesting comparison is given in Figure 1.6(page 28). In this example, again end–slicing on `x`, it turns out that the assignment to `y` can have no effect on the final value of `x`. The informal reasoning for this is that once we have done the assignment to `y`, the assignment to `x` can never be done again. The assignment to `y` can, therefore, have no effect

```
1    while i<0
     do
     begin
2    if c=3
      then
      begin
3      c:=4;
4      x:=5
      end;
5    i:=i+1
     end
```

Program $p_{1.2}$

```
while i<0
do
begin
 if c=3
 then
 begin

  x:=5
 end;
i:=i+1
end
```

Dataflow Minimal Slice

```
while i<0
do
begin
 if c=3
 then
 begin
  c:=4;
  x:=5
 end;
i:=i+1
end
```

Weiser's Algorithm

Figure 1.5: Illustration of the Dataflow Minimality Problem

on the final value of `x`. Weiser's algorithm includes the assignment to `y`.

In Figure 1.7(page 28), using Weiser's algorithm, slicing on `x` at the end of program $p_{1.7}$ gives the whole program whereas slicing using Weiser's algorithm on `x` at the end of program $p_{1.8}$ in Figure 1.8(page 28), gives the empty program. Clearly neither of these programs can have an effect on the final value of `x`. They either fail to terminate or leave `x` unchanged so in both cases, the dataflow minimal slice is the 'empty program'.

1.6 Organisation of this Thesis

The rest of the thesis is organised as follows:

- Chapter 2, **Slicing: Algorithms and Semantics** briefly surveys the main contributions to program slicing, focusing mainly on attempts to define the semantics preserved by slicing and related issues.
- Chapter 3, **Dataflow Dependencies** introduces the dataflow minimality problem and reasons are given for the lack of dataflow minimality of algorithms that use traditional

	Program $p_{1.6}$	Dataflow Minimal Slice	Weiser's Algorithm
1	`while x<z`	`while x<z`	`while x<z`
	`do`	`do`	`do`
	`begin`	`begin`	`begin`
2	`if x=3`	`if x=3`	`if x=3`
3	`then x:=x+y`	`then x:=x+y`	`then x:=x+y`
4	`else y:=y+1`		`else y:=y+1`
	`end`	`end`	`end`

Figure 1.6: End–slicing on x

	Program $p_{1.7}$	Dataflow Minimal Slice	Weiser's Algorithm
1	`while y>0`		`while y>0`
2	`do x:=x+1`		`do x:=x+1`

Figure 1.7: End–slicing on x

	Program $p_{1.8}$	Dataflow Minimal Slice	Weiser's Algorithm
1	`while y>0`		
2	`do y:=y+1`		

Figure 1.8: End–slicing on x

data and control dependence.

Four dependence relations all defined in terms of the semantics of programs are introduced.

1. *VD* and *TVD*, both binary relations between the variables of a program.
2. *LD* and *TLD* both binary relations between the variables and labels of a program.

Dataflow equivalence is formally defined and schemas [44, 78] are used for representing classes of programs with the same control flow graph.

For each of the above four dependence relations on programs, corresponding *dataflow dependencies* are defined:

1. *DVD* and *DTVD*, both binary relations on the set of variables of a schema.
2. *DLD* and *DTLD* both binary relations between the variables and labels of a schema.

These dataflow dependencies, (two of which are a form of slicing) are defined in such a way that *any* algorithm for computing them *must* be dataflow minimal.

- In Chapter 4, **The Semantics of Loop free Schemas**, the Symbolic Execution Tree: a structure used for performing dataflow analysis, is used. Symbolic execution trees are finite binary trees whose intermediate nodes are *symbolic predicates* and whose leaf nodes are *symbolic states* which map variable names to symbolic values. The semantics, $\mathcal{S}$, of loop-free schemas is defined as a mapping from loop–free schemas to symbolic execution trees. The chapter ends with an implementation of $\mathcal{S}$ in the functional language, Hope [6]. The input to this implementation is a representation of a schema s and the output is a representation of the symbolic execution tree, $\mathcal{S}[\![s]\!]$. This is the first stage in an algorithm for computing the dataflow dependencies introduced in Chapter 3.

- In Chapter 5, **The Soundness and Completeness of $\mathcal{S}$** it is shown how for each loop free schema, s, the symbolic execution tree, $\mathcal{S}[\![s]\!]$ characterises the set of all possible behaviours of all program in $[s]$. This characterisation provided by $\mathcal{S}$ is both sound and complete.

  - $\mathcal{S}$ is complete in the following sense:

    Given a loop–free schema s, and a program $p \in [s]$, and a state, σ, there exists exactly one path π of the symbolic execution tree, $\mathcal{S}[\![s]\!]$, that *corresponds* to the execution of p in state σ.

– $\mathcal{S}$ is sound in the following sense:

  For all paths π of the symbolic execution tree, $\mathcal{S}[\![s]\!]$ there exists a program $p \in [s]$, and a state, σ such that π corresponds to the execution of p in state σ.

- In Chapter 6, **Data and Control Dependence in Symbolic Execution Trees**, algorithms for computing *DTLD*, *DTVD*, *DLD* and *DVD* of loop-free schemas are given.

  For every loop–free schema s, these algorithms are defined in terms of its symbolic execution tree, $\mathcal{S}[\![s]\!]$.

  The fact that $\mathcal{S}[\![s]\!]$ properly characterises $[s]$ enables us to prove that the *DTLD* and *DTVD* algorithms for loop–free schemas are correct provided that the expression syntax of the underlying programming language is sufficiently rich.

  The algorithms for computing *DLD* and *DVD* are not proved correct.

  In order to compute each of the four dataflow dependencies of a loop–free schema s, two different versions of *data dependence* and four different versions of *control* dependence are defined. These forms of data and control dependence all operate on symbolic execution trees. Each of *DTLD*, *DTVD*, *DLD* and *DVD* is computed by applying the appropriate version of data and control dependence to $\mathcal{S}[\![s]\!]$.

  We show that *DLD* and *DTLD* can be thought of as special cases of *DVD* and *DTVD* respectively. *DTLD* can be computed by treating the labels as variables, computing the *DTVD*, and then restricting the final result to just the set of labels.

  This means that, in effect, the 'label' and 'variable' versions of each dependence above can be combined into a single dependence. This simplification implies that, in fact, just *one* form of data dependence and two forms of control dependence are all that is required in order to compute the four dataflow dependencies introduced in Chapter 3, when applied to loop–free schemas.

- In Chapter 7, **Computing Dataflow Dependencies of Schemas with Loops**, to compute the dataflow dependencies for schemas with loops, initially each loop is replaced by its *zeroth unfolding* and the dependence of this resulting loop–free schema is computed. These loop free schemas are then further unfolded and the dataflow dependencies are re–calculated. It is formally proved that this process will eventually terminate resulting with a loop–free schema whose dataflow dependence is the same as the program with loops with which we started.

Provided that we can recognise when further unfoldings will produce no further change in dependency, we have achieved dataflow minimal algorithms for computing the various dataflow dependencies introduced in this thesis.

- The thesis ends, (Chapter 8–**Conclusions**) with a summary of our findings together with indications for future research (Chapter 9 – **Future Work**). Due to their similarity to slicing, the applicability of the algorithms is only briefly mentioned. More of interest is the novelty of the approach and the potential improvements in the accuracy of dataflow analysis of programs that may result. Using different forms of control dependence, also defined in terms of symbolic execution trees it is probable that, using this approach, more accurate solutions to other related problems in dataflow analysis of programs may be found.

- Full listings of programs for $DTVD$ and $DTLD$, written in the functional programming language Hope [6] and example executions are included in the appendices. These implementations and examples can be tested using a web browser. (`http://p69-122.unl.ac.uk/~seb`)

Chapter 2

Slicing: Semantics and Algorithms

For a brief introduction to slicing and its applications, please refer to Chapter 1, Section 1.1, page 21.

2.1 Different Forms of Slice

There are many different definitions of program slices in the literature.

Slices can be *backward* or *forward* [59, 91], *static* or *dynamic* [5, 43, 68, 71], *intra–procedural* or *inter–procedural* [60, 59]. Slicing has been applied to programs with *arbitrary control flow* (goto statements) [7, 23, 2] and even concurrent programming languages like Ada [22, 97]. In a recent form of slicing called *amorphous* slicing [47, 14], slices are not necessarily produced by deleting statements and may not necessarily even be made from components of the original program being sliced. Amorphous slicing is so general, that it is, in effect a form of partial evaluation [37, 33, 17]. As will be seen, many forms of slicing are special cases of *dataflow analysis* [53, 92] i.e. working at the level of abstraction of *defined* and *referenced* variables, whereas others [87, 47] take more detailed information about expressions into account. Some definitions, for example *closure* slices [91] need not even be executable programs but just collections of labels corresponding to statements that affect the slicing criterion in some way.

2.1.1 Backward vs. Forward

A backward slice is the 'conventional one' [92] where it asked:

> Which statements affect the slicing criterion?

Forward slicing [60] is the converse of this[1]. The question asked in forward slicing is:

> Given a particular statement in a program, which other statements are affected by this particular statement's execution?

2.1.2 Static vs. Dynamic

A static slice is the conventional one where the slice is required to agree with the program being sliced in all initial states. Dynamic slicing [5, 43, 68, 67, 70, 72] involves executing the program in a particular initial state and using trace information to construct a slice relevant to this particular initial state.

There are variants of slicing in between the two extremes of static and dynamic[2], where some but not all properties of the initial state are known. These are known as conditioned slices [19] or constrained slices [35].

2.1.3 Intra–procedural vs Inter–procedural

Intra-procedural slicing means slicing programs which do not have procedures whereas inter–procedural [93, 59, 60, 68] slicing tackles the more complex problem of slicing programs where procedure definitions and calls are allowed[3].

2.1.4 Slicing Structured vs. Unstructured Programs

For many of these applications, particularly where maintenance problems are the primary motivation for slicing, the slicing algorithm must be capable of constructing slices from 'spaghetti' programs, written before the benefits of structured programming were fully appreciated[4]. The archetype of this unstructured programming style is the `goto` statement; all forms of 'jump' statement, such as `break` and `continue` can be regarded as special cases of the `goto` statement. Such programs are said to exhibit 'arbitrary control flow' and are considered to be 'unstructured'. The traditional program dependence graph approach [82] incorrectly fails

[1] In the main body of this thesis, only backward slicing is considered.

[2] In the main body of this thesis, only static slicing is considered. Although Section 2.6 in this chapter is an account of the dynamic slicing algorithms of Agrawal and Horgan [5].

[3] In the main body of this thesis, only intra–procedural slicing is considered.

[4] In the main body of this thesis, only slicing of structured programs is considered.

to include any goto statements in a slice. Various authors have suggested solutions to this problem [23, 2, 7, 48].

2.1.5 Dataflow vs. Non–Dataflow

In the non–dataflow analysis approach [73, 87], infeasible paths are detected by using 'cheeky rules' (defined in Section 2.2.3).

The fact that programs like:-

```
c:=1;
if c>0
then  x:=25
else  x:=z
```

and

```
c:=1;
x:=25
```

are semantically equivalent, can in certain circumstances be automated (although the general problem is clearly not solvable). As in the case of amorphous slicing [47], automatable or semi–automatable techniques involving the symbolic simplification of expressions will in turn lead to simpler semantically equivalent versions of the original program and hence to thinner slices than those produced by dataflow analysis alone[5]. Another non dataflow approach is *parametric program slicing* [35, 36] , where slices are constructed using a term-rewriting system, which can use arbitrary rewrites which preserve a property of the syntax using origin tracking [36]. The re-write rules can be cheeky (see Section 2.2.3) because they can involve information about expressions other than their referenced variable sets.

Many examples of slicing are combinations of the categories above. For example the work of Kamkar [66] produces backward, dynamic, inter–procedural slices. Weiser's original work described backward, static, intra–procedural slicing although he also gave an algorithm for backward, static, inter–procedural slicing.

2.2 Weiser's Work

Weiser's main achievement was to produce an algorithm for producing backward static intra–procedural slices. His algorithm is an example of **dataflow analysis**, a central theme of this thesis. Its input is the control flow graph of the program being sliced.

[5] Such techniques lie outside the scope of this thesis.

2.2.1 Control Flow Graphs

A control flow graph is an abstract representation of a program. For example, consider the program in Figure 2.1(page 36).

```
1    while i<2
     do
     begin
2       if c=2
        then
        begin
3          c:=y;
4          x:=25
        end;
5       i:=i+1
     end
```

Figure 2.1: Program $p_{2.1}$

The control flow graph of a program has nodes which have been labelled with the defined and referenced variables at each node. The control flow graph, $G_{2.1}$, of program $p_{2.1}$ in Figure 2.1 is given in Figure 2.2(page 37).

The reason that a control flow graph is more abstract than a program is that whereas programs have expressions on the right hand side of assignments and as guards of conditionals and loops, control flow graphs have sets of variable names. This set is the set of variables **referenced** by the corresponding expression: i.e. the set of variable names explicitly mentioned in the corresponding expression. The **defined variables** are the ones occurring on the left hand side of assignment statements.

2.2.2 Dataflow Analysis

Weiser defined *Dataflow Analysis* [53] to be the analysis of a program's control flow graph. All we are allowed to take advantage of in such analysis are the sets of *defined* and *referenced* variables at each node of the control flow graph.

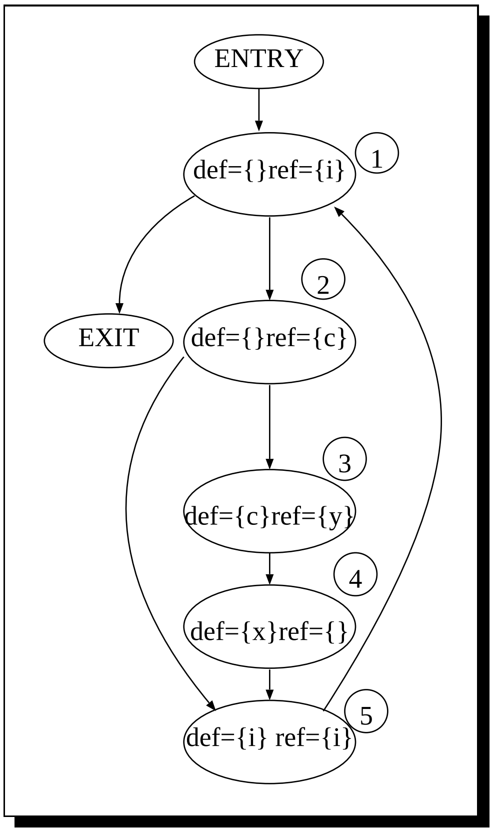


Figure 2.2: $G_{2.2}$: The control flow graph of $p_{2.1}$

2.2.3 Inherent Inaccuracies in Dataflow Analysis

Since they reference the same set of variables, dataflow analysis cannot distinguish between the expressions `2*x` and `x-x`. All programs differing only in this way would be treated identically. The fact that `x-x` can be replaced by 0, an expression that no longer references `x`, is an example of what we call a *cheeky rule* since finding the sets of variables upon which expressions *depend* (rather than simply *mention*) is not computable. Applying cheeky rules is not part of dataflow analysis.

Another way of thinking of it is that dataflow analysis is performed not on programs but on control flow graphs. All the 'dataflow analyser' is presented with, therefore, are sets of variables in place of expressions. The ability to apply cheeky rules is thus removed.

2.2.4 Traditional Dependence

Weiser's algorithm and most subsequent work on program dependence uses two relations [53] between the nodes of a program's control flow graph. These are

1. Data Dependence(D)
2. Control Dependence(C)

2.2.5 Data Dependence

Node n_2 is *data dependent* on node n_1 if there is a variable v referenced in n_2 which is defined in n_1 and there is a path from n_1 to n_2 with no intervening assignments to v. We write $n_1\ D\ n_2$ to mean n_2 is data dependent on n_1.

Examples of Data Dependence

Consider:

```
n1        x:=y;
.         ...
n2        z:=x
```

If there are no intervening assignments to `x` then `n2` is data dependent on `n1` since the value of `x` at `n2` is 'affected by' the value of `y` at `n1`. Similarly, consider:

```
          while b do
          begin
A         ...
n2        z:=x;
.         .
.         .
.         .
n1:       x:=y;
B         ...
          end
```

Again, if there are no assignments to `x` in the portions of code labelled `A` and `B` then `n2` is data dependent on `n1` since the value of `x` at `n2` is 'affected by' the value of `y` at `n1`. This is an example of a *loop carried* data–dependence.

2.2.6 Control Dependence

Control Dependence is harder to define. Informally, the execution of a predicate node 'controls' the execution of other nodes in the control flow graph by determining whether or not control will definitely pass to these nodes or not. For each predicate node, b, the set of nodes that depend on the outcome of b in this way are termed *the controlled nodes* of b. More formally, control dependence is defined in terms of post–dominance:

Definition 2.2.1 (Post–dominance)
A node i is *post–dominated* by a node j if all paths from i to EXIT pass through j.

Definition 2.2.2
A node j is control dependent on node i if and only if

1. There exists a path π from i to j such that for all u in π with $u \neq i$ and $u \neq j$, u is post–dominated by j
   and
2. i is not post–dominated by j.

A predicate node b is a node like nodes 1 and 2 in Figure 2.2(page 37). All predicate nodes have two arcs leading from them corresponding to true and false. Predicate node b controls n if all paths from π to the exit starting with one *do* contain n and there exists a path from π to the exit starting with the other arc that *does not* contain n. In a block structured language, like the one being considered in this thesis, the set of nodes controlled by a predicate b can be defined syntactically. If we associate each assignment statement with its corresponding node in the control flow graph and each loop and conditional with the node in the control flow graph corresponding to its predicate, then the set of nodes controlled by b are simply the nodes corresponding, in the way just described, to the statements that appear at depth one in the abstract syntax tree of the statement whose predicate is b.

For unstructured languages, the calculation of controlled nodes can be achieved using the algorithm of Ferrante, Ottenstein and Warren [34].

2.2.7 Weiser's Algorithm

In Weiser's original thesis [92] and Tip's later exposition [89] it is shown how 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 entails computing sets of 'relevant variables' for each node in the control flow graph.

Directly Relevant Variables

Suppose a slice is to be constructed for the slicing criterion $C = (V, n)$. First, the *directly relevant variables* of node i, $R^0_C(i)$, are defined inductively as follows:-

1. The set of directly relevant variables at the slice node, n, is simply the slice set, V.
2. The set of directly relevant variables at every other node i, is defined in terms of the set of directly relevant variables of all nodes j leading directly from i to j (written $i \rightarrow_{CFG} j$) in the control flow graph. $R^0_C(i)$ contains all the variables v such that either
   (a) $v \in R^0_C(j)$ and $v \notin \mathit{def}(i)$ or
   (b) $v \in \mathit{ref}(i)$ and $\mathit{def}(i) \cap R^0_C(j) \neq \emptyset$.

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

Directly Relevant Statements

In terms of the directly relevant variables, a set of *directly relevant statements* S^0_C is defined:-

$$S^0_C = \{i \mid \exists j \text{ such that } i \rightarrow_{CFG} j \text{ and } \mathit{def}(i) \cap R^0_C(j) \neq \emptyset\}$$

This completes the first iteration of Weiser's algorithm.

Indirectly Relevant Variables

The subsequent iterations of Weiser's algorithm calculate the *indirectly relevant variables*, R^K_C where $K \geq 0$.

In calculating the indirectly relevant variables, control dependence is taken into account.

$$R^{K+1}_C(i) = R^K_C(i) \cup \bigcup_{b \in B^K_C} R^0_{(\mathit{ref}(b),b)}(i)$$

where

$$\mathrm{B}_C^K = \{b \mid \exists i \in S_C^K \text{ such that } b \text{ controls } i\}$$

B_C^K is the set of all predicate nodes that *control* a statement in S_C^K.

Indirectly Relevant Statements

Adding the predicate nodes to S_C^K includes further indirectly relevant statements in the slice:-

$$S_C^{K+1} = B_C^K \cup \{i \mid \exists j \text{ such that } i \rightarrow_{CFG} j \text{ and } def(i) \cap R_C^{K+1}(j) \neq \emptyset\}$$

As Tip [89] states, this process will eventually terminate since S_C^K and R_C^K are non–decreasing subsets of the program's variables.

Weiser proves, in his thesis(theorem 10), that his algorithm produces slices according to his semantic definition of a slice.

2.3 A Parallel version of Weiser's Algorithm

As has just been shown, Weiser's algorithm is fairly complicated to express using conventional techniques. In this section, we show that Weiser's algorithm [92] can be expressed more elegantly using a parallel algorithm [28].

Parallel algorithms have the potential for being 'faster' than their sequential counterparts, since, as their name suggests, the work can be shared by many computing agents all executing at the same time.

The reason why parallel algorithms are of interest here, however, is not a question of improved efficiency, but one of improved 'expressibility'.

Often, problems can be expressed more generally and more clearly concurrently rather than sequentially because the problem itself may have inherent concurrent aspects. A sequential algorithm is just a 'special case' of a parallel one. Sequential notations often force the programmer to impose an unnecessarily strict order of computation on his algorithm.

The simplest example of this is demonstrated in 'sequential' versus 'parallel' assignment [32]. Suppose a programmer wants to write a program that leaves variable x having the value 7 and y the value 9. He might choose `x:=7;y:=9` or `y:=9;x:=7` clearly the order of execution of these two assignment statements is immaterial. If we had a notation for expressing 'parallel assignment' the the programmer could just write `(x,y):=(7,9)`. The fact that having such parallel constructs leads to simpler algorithms can be seen in a simple example where the programmer wishes to swap the values of variables x and y. As every programmer knows, to do this in a sequential language, a temporary variable is usually used. In the case of parallel assignment we would simply write `(x,y):=(y,x)`.

Programs written in functional languages are another example. These languages have no concept of sequentialisation of statements, and thus are inherently concurrent.

Much work [64, 65, 55] has been done in expressing parallel algorithms using networks of communicating processes all acting concurrently. In general the behaviour of each process in the network is very simple. The algorithmic power is obtained by the interaction and co-operation of these simple processes. In [28], a similar approach to [1] is adopted, where each process can be defined by a simple function on streams of messages and the topology of the process network is defined by the way that the individual functions are composed.

Weiser's algorithm is expressed as a network of processes each of whose behaviours is very simple. Interestingly, the topology of the network is *exactly* the same as the topology of the control flow graph of the program being sliced. Each process in the network, therefore, corresponds to a node of the control flow graph and its behaviour can be defined very simply in terms of properties of this node.

Communication between processes is in the **reverse direction** of the arrows in the control flow graph. So output channels correspond to arcs *entering* a node and input channels correspond to arcs *leaving* a node.

2.3.1 Process Behaviour

Each process repeatedly sends and receives messages that are sets consisting of variable names and node numbers. The behaviour of each process, i, depends precisely on the following information, derived directly from the control flow graph of the program being sliced:-

i	The number of the corresponding node of the control flow graph
$ref(i)$	The set of variables referenced by i
$def(i)$	The set of variables defined by i
$C(i)$	The set of nodes controlled by i

Processes with more than one input correspond to predicate nodes. This work is concerned with side-effect free languages, so all such processes will have $def(i) = \emptyset$. Conversely, processes with only one input do not correspond to predicate nodes, and therefore, by definition, they control no other nodes and so have $C(i) = \emptyset$. The behaviour of each process, i, defined in functional notation is:-

$$
\begin{array}{l}
F_i : \text{set(name)} \rightarrow \text{set(name)}; \\
\begin{array}{rl}
F_i(S) = & \text{if } S \cap (def(i) \cup \mathrm{C}(i)) \neq \emptyset \\
& \text{then } (S - def(i)) \cup ref(i) \cup \{i\} \\
& \text{else } S
\end{array}
\end{array}
$$

This is interpreted as follows:-
If the input, S, to process i, has any elements in common with the defined variables of i or with the controlled nodes of i then the process, i, outputs the set consisting of :-

1. all its input variables (elements of S) that it does not define,

2. all variables that it references,

3. its node number, i.

On the other hand, if S has no elements in common with the defined variables or controlled nodes of i then the process i merely outputs S.

The process i then repeats this action, waiting for the next input message.

2.3.2 Starting Network Communication

In order to construct a slice for the criterion (V, n), network communication is initiated by outputting[6] the message V from process n. Messages will be then passed around the network

[6] According to Weiser's original definition of a static slice [92], a slice constructed for the slicing criterion (V, n), need not contain the node n. Korel and Laski [71] argue that the omission of the slice node is unfortunate; a programmer finds the absence of the slice node confusing when attempting to relate the slice to the original. In order to include the slice node in a slice constructed by the parallel slicing algorithm, process communication should be initiated by outputting, not just the set of variables V, but also the node identifier, n.

until it eventually 'stabilises' when no new messages arise from any node.

2.3.3 Constructing the Slice

Once stabilised, the slice is computed by observing what has been output by each node: Node i should be included in the slice if and only if process i has output its node identifier i. The slice of a program computed by this algorithm can be found by including the set of nodes whose identifiers are input to the entry node of the control flow graph, because the entry node is reachable via every node in the reverse control flow graph and thus messages output by all nodes will eventually reach the entry process.

Alternatively, we imagine that all the nodes have lights on them. A node 'lights up' if it ouptuts its own identifier. The slice is simply the set of 'lit up' nodes.

Definition 2.3.1 above, should be thought of as a 'specification' of process behaviour rather than an 'implementation'. The important aspect of the definition is that each process should be thought of as a function from the union of all its inputs to the union of all its outputs.

2.3.4 Example Execution of the Parallel Algorithm

As is the convention, arcs entering a node, i, represent inputs to process, i, and arcs leaving i represent outputs from process, i. When a process outputs a message, it shall mean that the message is output on all output channels. Let the slicing criterion be ({c},7). The program to be sliced is shown in figure 2.3.

```
1    a:=0;
2    while s<t
3    do begin if t=4
4             then c:=t;
5             s:=2;
6             c:=t+7;
7             t:=a+4
        end
```

Figure 2.3: The program to be sliced

The reverse control flow graph of the program to be sliced is shown in figure 2.4:-

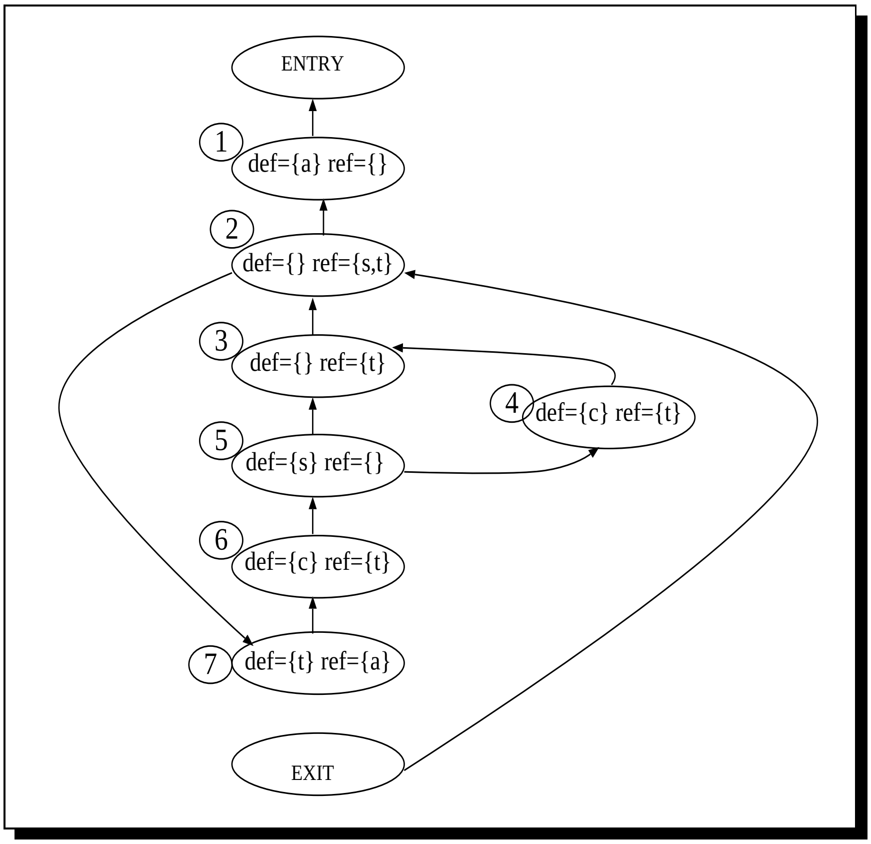


Figure 2.4: The reverse control flow graph obtained from the initial program

The process network obtained from the reverse control flow graph is shown in figure 2.5. A slice is to be constructed for the criterion ({c},7), so process communication is initiated by outputting {c} from node 7.

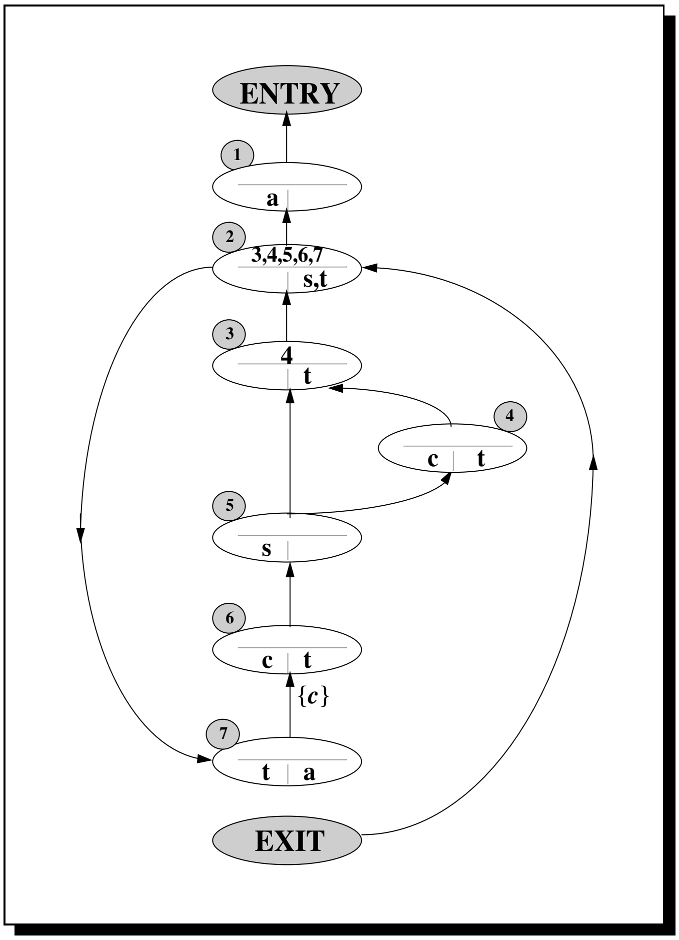


Figure 2.5: Initial state of the process network

To show the progression of the state of the system, the arcs are labelled in the reverse control flow graph with the messages communicated by the relevant processes during execution. New messages communicated at each stage are labelled in bold typeface.

Processes are drawn like this:

Where C is the set of controlled nodes, DEF is the set of defined variables and REF is the set of referenced variables of each process.

After receiving {c} through its input channel, process six outputs {t,6}. Processes five, four and three all eventually receive {t,6} which they simply output because {t,6} is disjoint from the defined variables of these processes.

The resulting state is shown in figure 2.6.

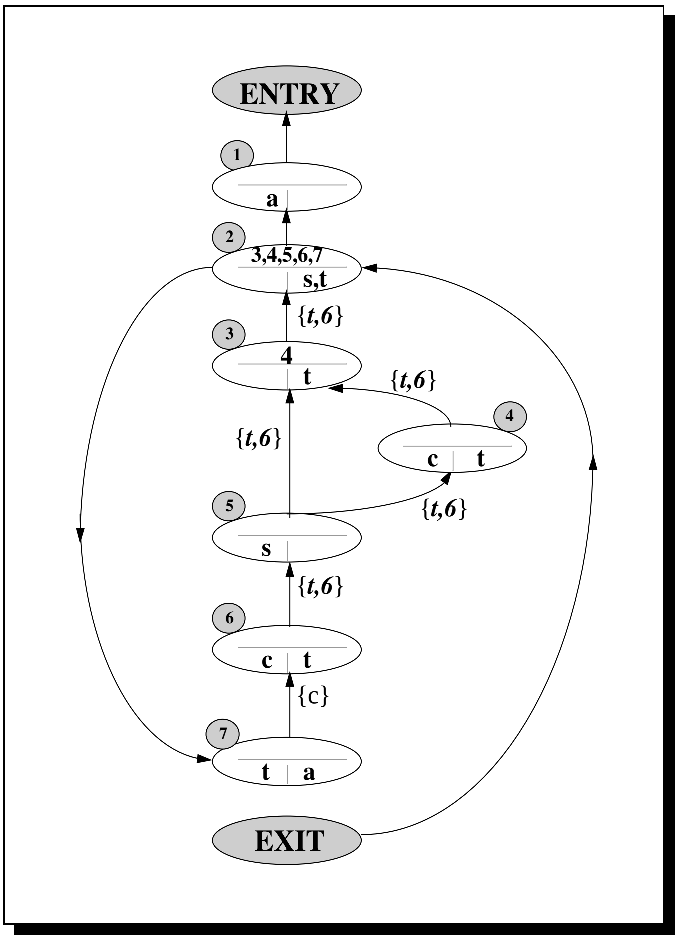


Figure 2.6: The state just before process two outputs its first message

When {t,6} is input to process two, it causes process two to output {s,t,2,6} to processes one and seven. This is an instance of a process responding to an input containing the identifier of a node that it controls. Process two will therefore output a message including its own node identifier, representing the fact that node two will also be included in the final slice.

The resulting state is shown in figure 2.7.

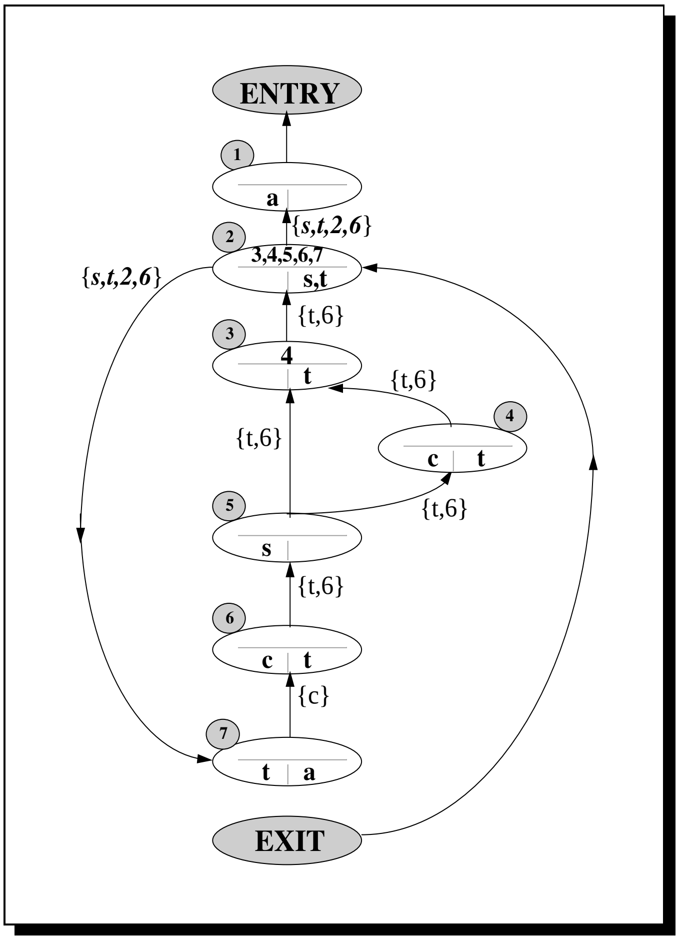


Figure 2.7: The state just after process two has output its first message

The message {`s,t,2,6`} passes through process one to the `ENTRY` node. On receiving {`s,t,2,6`}, process seven outputs the message {`s,a,2,6,7`} because it defines `t`. This output message passes unaffected through process six.

When process five receives {`s,a,2,7`} it outputs {`a,2,5,7`}.

The resulting state is shown in figure 2.8.

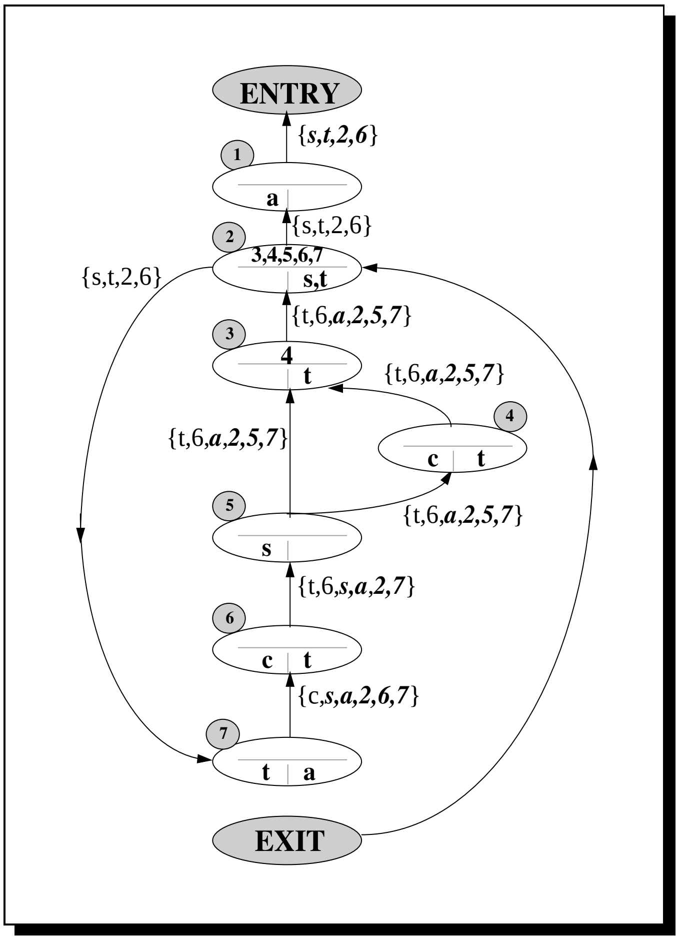


Figure 2.8: The state after one more pass around the loop

Continuing process communication passes the extra messages in the network to all reachable nodes, but causes no new messages to be introduced into the system. Finally the network terminates in the state shown in figure 2.9.

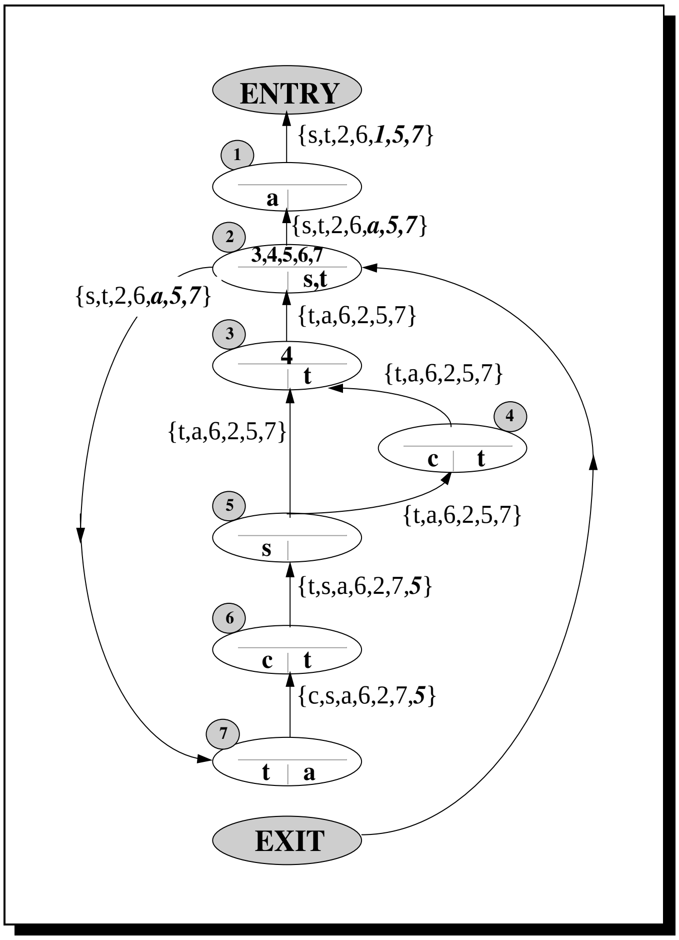


Figure 2.9: The final state of the process network

From the final state of the network the slice of the original program is constructed by including those statements and predicates whose node identifiers, $\{1, 2, 5, 6, 7\}$, have reached the `ENTRY` node.

```
1     a:=0;
2     while s<t
3     do begin if t=4
4              then c:=t;
5              s:=2;
6              c:=t+7;
7              t:=a+4
         end
```

```
a:=0;
while s<t
do begin

         s:=2;
         c:=t+7;
         t:=a+4
   end
```

Figure 2.10: The original program and its slice

A proof of the equivalence of Weiser's and this algorithm is included in appendix C.

2.4 Weiser Slices

In this section we describe *what* Weiser's algorithm produces, in terms of a program's control flow graph, rather than *how* it produces it. Suppose we wish to construct the Weiser slice for program p with slicing criterion (V, n). First we insert a node with referenced set V into the desired place in the control flow graph of p. For example, if we were slicing at the end of the program we would place this extra node after the exit node. Call this node n'. Let

$$F = D \cup C$$

i.e. F is the union of the data and control dependence relations. The Weiser slice is 'more or less' the set of nodes consisting of the transitive closure F^* of F applied to node n'. To be precise, it is

$$(F^* \circ D) \cup D.$$

In other words it is all the maplets in the transitive closure that start with a data dependence.

Example

Suppose we wished to slice at the end of program $p_{2.1}$ in figure 2.1 with respect to variable x at the end of the program. We first add the node with referenced variables $\{x\}$ at the appropriate place in the control flow graph of $p_{2.1}$ to get the augmented control flow graph in figure 2.11. In this case:

$$C = \{2 \mapsto 1, 3 \mapsto 2, 4 \mapsto 2\}$$

$$D = \{1 \mapsto 5, 2 \mapsto 3, 6 \mapsto 4\}$$

so

$$F^* = \left\{ \begin{array}{l} 1 \mapsto 5 \\ 2 \mapsto 1 \\ 2 \mapsto 2 \\ 2 \mapsto 3 \\ 2 \mapsto 5 \\ 3 \mapsto 1 \\ 3 \mapsto 2 \\ 3 \mapsto 3 \\ 3 \mapsto 5 \\ 4 \mapsto 1 \\ 4 \mapsto 2 \\ 4 \mapsto 3 \\ 4 \mapsto 5 \\ 6 \mapsto 1 \\ 6 \mapsto 2 \\ 6 \mapsto 3 \\ 6 \mapsto 4 \\ 6 \mapsto 5 \end{array} \right\}$$

and in this case

$$((F^* \circ D) \cup D)6 = \{1, 2, 3, 4, 5\}.$$

2.4.1 Slicing using Program Dependence Graphs

Program dependence graphs were introduced by Ferrante et al [34]. The program dependence graph for a program, P, is a directed graph whose nodes are connected by several different kinds of edge. For example, the program dependence graph, $G_{2.4.1}$, of program, $p_{2.1}$, is given in Figure 2.4.1(page 54). The nodes are essentially the same nodes that occur in the control flow graph. The arcs of a program dependence graph that are relevant for slicing are the

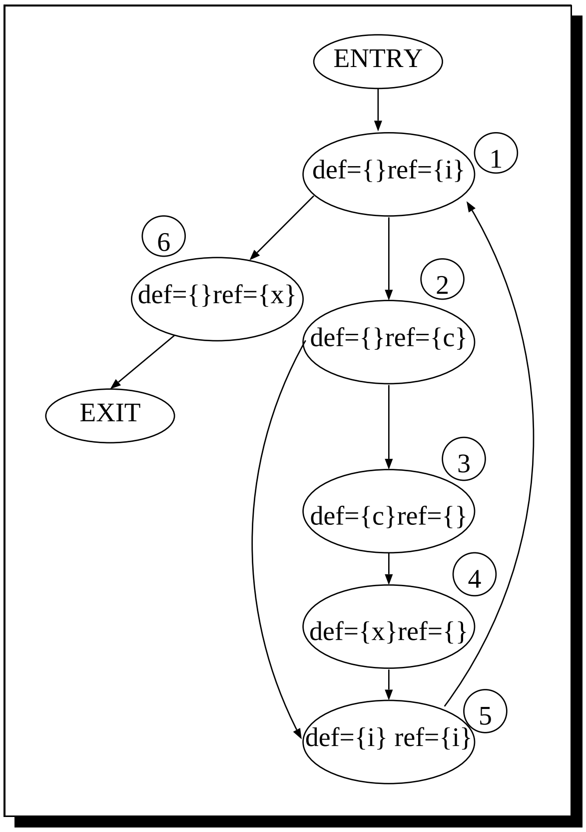


Figure 2.11: $G_{2.11}$: The augmented control flow graph of $p_{2.1}$

control dependence arc and the *data dependence arc*. There is a control dependence arc from node n_1 to node n_2 if and only if n_2 is control dependent on n_1 and there is a data dependence arc from node n_1 to node n_2 if and only if n_2 is data dependent n_1.

Ottenstein and Ottenstein [82] showed how slicing could be done using a program dependence graph. Most current implementations [3, 58] of slicing algorithms use the program dependence graph. The reason for this is one of efficiency. In effect, the process of computing the program dependence graph is performing the 'hard work' in computing the slices for n slicing criteria where n is the number of nodes. As a result of this 'pre–processing', any *particular* slice corresponding to one of these n criteria can be computed in linear time.

The slices produced are usually, but not always, exactly the same as those produced by Weiser's algorithm. The differences between slicing using Weiser's algorithm and program dependence graphs can be enumerated as follows:-

1. The slice produced by the program dependence graph approach is *exactly* the transitive closure of the union of the data dependence and control dependence relations. This means that the program dependence graph approach sometimes produces bigger slices than Weiser's algorithm.

2. Using the program dependence graph approach, slicing cannot be done with arbitrary slicing criteria. There is no concept of adding nodes at the slice point. A slice can only be constructed using existing nodes. So slicing at node n can only be done with the referenced variables of node n.

To calculate a PDG–slice at a node n we simply trace back along all these arcs. We include in the slice every node that we reach. Clearly this will simply produce

$$(D \cup C)^* n.$$

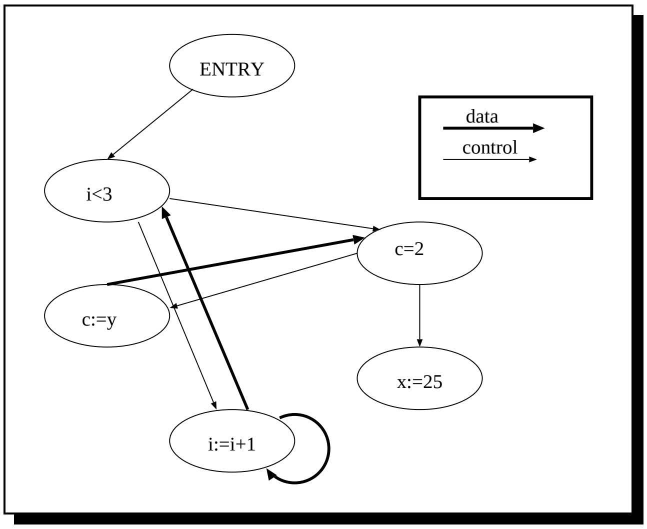


Figure 2.12: $G_{2.4.1}$: The program dependence graph of $p_{2.1}$

2.5 The Semantics of Slicing

An essential issue in program slicing is to define what it means for two programs to behave the same with respect to a slicing criterion. i.e. What *semantic relationship* must exist between a program and its slice in order that the slice is considered valid.

2.5.1 Weiser's Semantic Definition of Valid Slices

Weiser defined the semantic relationship that must exist between a program and its slice in terms of state trajectories:

State Trajectories

A state trajectory is a sequence of label, state pairs (l_i, σ_i) where σ_i represents the state immediately before executing the statement labelled l_i.

Definition 2.5.1 (Weiser Slices)

A slice s of a program p on a slicing criterion $c = (V, i)$ is any executable program with the following property. Whenever p halts on an input I with a state trajectory T the s also halts on input I with state trajectory T' with

$$\mathrm{Proj}_c(T) = \mathrm{Proj}_c(T')$$

$\mathrm{Proj}_c(T)$ is obtained first by deleting all elements of T whose label component is not i and then, by restricting the state components to V[7].

2.5.2 End Slicing

When slicing at the end of the program[8], the trajectories will all be of length one (since the 'exit' statement is executed only once). This gives rise to simplified form of slicing called *end slicing*.

Definition 2.5.2 (Weiser's Definition of an End–Slice)

p' is an end slice of p with respect to a set of variables V if whenever p terminates so does p' with the same final values of the variables in V.

[7] This is a slight simplification of the true picture since we are assuming that i is in the slice of p with respect to c. A more complicated definition involving 'nearest successors' is required if i is not in the slice

[8] This thesis solely concerns end–slicing.

2.5.3 Slicing and non–Termination

Using Weiser's definition, *every* program is semantically a valid slice of p, for non–terminating programs p. Weiser, himself, noticed that using his algorithm, there is no guarantee that a slice will fail to halt whenever the original program fails to halt. In other words, the Weiser slice of a program may terminate in some states where the original did not. So the slices produced by Weiser's Algorithm do not preserve a projection of the standard semantics [88] of programs.

2.5.4 The Semantics of the PDG approach

Horwitz et al. [56] show that a program dependence graph (where the nodes contain the atomic statements and not just the defined and referenced variables) is an adequate structure for representing a program's execution behaviour in the sense that two program's with the same program dependence graph have the same standard semantics. Reps and Yang [83] prove that the program dependence graph approach to slicing preserves Weiser's semantics i.e. it was shown that for any initial state where the original program terminates then the slice also terminates with the same sequence of values for each element of the slice. The converse is not true i.e. in some states the slice may terminate when the original program does not. Cartwright and Felleisen [21] define a *lazy semantics* of programs which they show is preserved by dataflow slicing algorithms like Weiser's Algorithm [92] and the program dependence graph approach [82]. Before their work is discussed, a short introduction to standard denotational semantics is required.

2.5.5 Standard Semantics

Definition 2.5.3 (state)

In denotational semantics [88], a state, $\sigma \in \Sigma$, is a mapping from variables to values from a set V.

$$\sigma \in \Sigma = \mathit{variables} \nrightarrow \quad V$$

For example the function

$$\sigma = \left\{ \begin{array}{c} x \mapsto 5 \\ y \mapsto 1 \\ z \mapsto 2 \end{array} \right\}$$

represents the state where variable x has the value 1, variable y has the value 1 and z the value 2.

The meaning of a program is given by a function from states to states [88].

$$\mathcal{M} : P \rightarrow \Sigma \nrightarrow \Sigma$$

Where P is the set of all programs. Given an initial state σ, the final state reached after executing p starting in state σ is thus written $\mathcal{M}[\![p]\!]\sigma$. If the program p does not terminate when started in state σ, then $\mathcal{M}[\![p]\!]\sigma$ has the special value $\perp$, pronounced 'bottom'. In standard semantics, in the bottom state all variables are deemed to have the value $\perp$. So the bottom state is the function that maps every variable name to $\perp$.

The final value of variable x after executing p in initial state σ is thus written[9] $\mathcal{M}[\![p]\!]\sigma\ x$.

Ordering on States

In the standard semantics, the ordering on states is such that two distinct non $\perp$ states are incomparable and $\perp$ is less than every state. The reason an ordering is required is that the meaning of loop is defined to be the least fixed point of a function. The ordering expresses the sense in which the fixed point is the least.

Evaluating Expressions

The meaning of an expression e is given by the function $\mathcal{E}$ which evaluates an expression in a state to give a value.

$$\mathcal{E} : expressions \rightarrow \Sigma \nrightarrow V$$

[9] Function application associates to the left. i.e. $f\ g\ h\ x$ means $((fg)h)x)$

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 even expressions like 3 that do not reference any variables.

Assignment Statements

The meaning of an assignment statement is the conventional [88]

$$\mathcal{M}[\![x := \ e]\!] = \lambda\sigma.\sigma[x \leftarrow \mathcal{E}[\![e]\!]\sigma]$$

$f[x \leftarrow y]$ is the function that is the same as f except that x is mapped to y. This is known as the update operation. The new state after doing an assignment statement `x:=e` is the same as the old except that x gets mapped to the value of the expression e evaluated in the old state.

Sequences of Statements

The meaning of s_1 followed by s_2 is the composition of the meaning of s_2 with the meaning of s_1.

$$\mathcal{M}[\![s_1;s_2]\!] = \lambda\sigma.\mathcal{M}[\![s2]\!](\mathcal{M}[\![s1]\!]\sigma)$$

Conditionals

For conditionals, the guard is evaluated in the current state. If it is true then the result is the final state after executing the `then` part, otherwise it is the final state after executing the `else` part.

$$\mathcal{M}[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2]\!] = \lambda\sigma.\mathcal{E}[\![b]\!]\sigma \rightarrow \mathcal{M}[\![s_1]\!]\sigma, \mathcal{M}[\![s_2]\!]\sigma$$

The notation $[a \rightarrow b, c]$ corresponds to the function which if a is true gives b, if a is false it gives c and if a is $\perp$ it gives $\perp$.

Loops

The meaning of a while loop $\mathcal{M}[\![\texttt{while}\ b\ \texttt{do}\ s]\!]$ is defined to be the least fixed point of the function

$$\lambda\omega\sigma\mathcal{E}[\![b]\!]\sigma \rightarrow \omega(\mathcal{M}[\![s]\!]\sigma), \sigma$$

which is shorthand for:-

$$\lambda\omega\sigma \begin{cases} \omega(\mathcal{M}[\![s]\!]\sigma) & \text{if } \mathcal{E}[\![b]\!]\sigma \\ \sigma & \text{otherwise} \end{cases}$$

The type of this function is:- (state $\rightarrow$ state) $\rightarrow$ (state $\rightarrow$ state). So its least fixed point is a state to state function.

Meaning of Loops(Example)

Consider

$$\mathcal{M}[\![\texttt{while}\ true\ \texttt{do}\ x{:=}1]\!]$$

is evaluated. By definition, it is the least fixed point of

$$F = \lambda\omega\sigma \begin{cases} \omega(\mathcal{M}[\![x{:=}1]\!]\sigma) & \text{if } \mathcal{E}[\![true]\!]\sigma \\ \sigma & \text{otherwise} \end{cases}$$

We are looking for the least ω such that

$$\omega = \lambda\sigma \begin{cases} \omega(\mathcal{M}[\![x{:=}1]\!]\sigma) & \text{if } \mathcal{E}[\![true]\!]\sigma \\ \sigma & \text{otherwise} \end{cases}$$

$\mathcal{E}$ is strict so

$$\mathcal{E}[\![true]\!]\sigma = \begin{cases} \perp & \text{if } \sigma = \perp \\ true & \text{otherwise .} \end{cases}$$

From this it is clear that $\lambda\sigma \cdot \perp$ is a fixed point of F and hence the least.

This means the program `while` $true$ `do` x:=1 will fail to terminate, whatever its initial state.

Example

Now consider,

$$\mathcal{M}[\![\texttt{while}\ true\ \texttt{do}\ x{:=}1; x{:=}1]\!].$$

Using the sequence rule this gives

$$\mathcal{M}[\![x{:=}1]\!](\perp)$$

which by the assignment rule gives:-

$$\perp[x \leftarrow \mathcal{E}[\![1]\!]\perp]$$

which is $\perp$ since $\mathcal{E}[\![1]\!]\perp = \perp$ since $\mathcal{E}$ is strict in both its arguments.

2.5.6 Lazy Semantics

Lazy semantics is a term usually applied to functional languages [42]. An interpreter that performs lazy evaluation will result in some programs terminating that would not do so if the opposite form of evaluation called *eager* evaluation were used. The reason this happens is that in lazy evaluation, when applying a function to some arguments, the arguments are only evaluated if their value is need. In eager evaluation, on the other hand, the arguments are always evaluated before the function is applied. If evaluating an argument, therefore, leads to non-termination, and this argument is not needed, then eager evaluation will lead to non–termination but lazy evaluation may not. An example is given in the functional language Hope [6].

```
dec f,g:num -> num;
g(n) <= g(n);
f(n) <= 1;
```

Eager evaluation of the term `f(g(0))` will produce non termination ($\perp$) whereas lazy evaluation will produce 1.

Now `g(0)` produces $\perp$ in both lazy and eager evaluation. So in lazy evaluation `f(g(0))` $=$ `f(`$\perp$`)` which evaluates to 1. Using lazy evaluation, function application is not strict, whereas using eager evaluation function application is strict.

Unlike in functional languages, it does not make sense to have a lazy interpreter for imperative languages, since in imperative languages we are interested in intermediate computation and not just the final result, however it is still possible to define a lazy semantics of imperative languages. (The standard semantics is eager.)

For imperative programs, the state in lazy semantics can map some variables to $\perp$ and others to proper (non–$\perp$) values. The ordering on states is the same however. Evaluating expressions in the state where some of the variables get mapped to $\perp$ can produce a non–$\perp$ value if none of the variables needed to evaluate the expression are mapped to $\perp$. i.e. $\mathcal{E}$ is non–strict.

As will be shown, even in lazy semantics, an infinite loop results in the same state (namely, $\lambda x.\perp$). The difference is that in lazy semantics, subsequent assignments can cause this state to be 'recovered'.

Consider again, the program `while` true `do` x:=1.

Using lazy semantics, we are looking for the least ω such that

$$\omega = \lambda\sigma \begin{cases} \omega(\mathcal{M}_{lazy}[\![x{:=}1]\!]\sigma) & \text{if } \mathcal{E}[\![true]\!]\sigma \\ \sigma & \text{otherwise} \end{cases}$$

In lazy semantics, $\mathcal{E}[\![true]\!]\sigma = true$ for all σ.

So we are looking for the least ω such that

$$\omega = \lambda\sigma\omega(\mathcal{M}_{lazy}[\![x{:=}1]\!]\sigma)$$

which is still $\lambda\sigma \cdot \perp$

but, now consider,

$$\mathcal{M}_{lazy}[\![\texttt{while}\ true\ \texttt{do}\ x{:=}1; x{:=}1]\!].$$

Again, as above, this gives

$$\perp[x \leftarrow \mathcal{E}[\![1]\!]\perp]$$

but because $\mathcal{E}$ is non–strict, this gives

$$\perp[x \leftarrow 1].$$

This is the state that maps every variable to $\perp$ except `x` that is mapped to 1, since $\mathcal{E}$ is non–strict in both its arguments. So in other words the program has 'recovered from' the infinite loop. Projecting the lazy meaning of the program above onto the variable `x` gives the value 1. Although in 'normal execution' the program will not terminate.

The fact that slicing preserves lazy semantics has the consequence that slicing is allowed to *introduce termination.* While lazy semantics is the norm for functional programming languages, it is not normally associated with the meaning of imperative programs, for which slicing is, almost exclusively, applied[10].

Consider the example program in figure 2.13. A static slice constructed with respect to (`x`,3) will (conventionally) contain line 3 alone. The fact that line four will never be executed when `y` is initially greater than 0 is of no consequence. In the lazy semantics of this program the final value of the variable `x` is 1, whatever the initial state.

2.5.7 Statement Minimal Slices

Clearly, by definition, every program is a slice of itself and in general the slice of a program is not unique, since we can add statements of p to p' which have no effect on the slicing criterion and still have a slice. Since every program is a slice of itself, a correct but useless

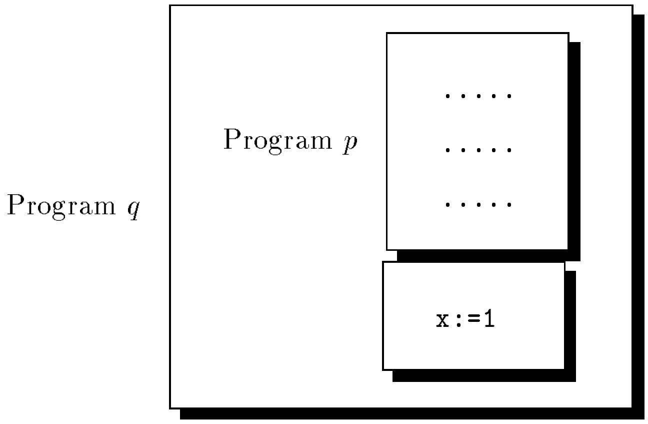
[10] Slicing has also been applied to functional style notations [96].

```
while(y<0)
do  y:=y+1;
x:=1;
```

Figure 2.13: Non-Termination Preservation

slicing algorithm would be one that simply performed the identity function on programs. It is clearly desirable to have an algorithm that produces slices that are 'as small as possible'.

A statement minimal slice is one where as many statements as possible have been deleted. Weiser showed that it is not possible to write an algorithm for finding statement minimal slices for arbitrary programs. He did this by showing that if we could find statement minimal slices we could also solve the halting problem. To find whether a given program p halts, simply compute the end slice of x with respect to the program q:-



If p does not terminate, the statement minimal end slice of program q will give the empty program and if it does terminate it will give

```
x:=1
```

.

2.5.8 Dataflow Minimality Problem

The idea is central to this thesis. An account of the dataflow minimality is given in Chapter 1 and is not repeated here.

2.5.9 Venkatesh's Work

The major aim of the work by Venkatesh [91] is to separate definitions of slices from the algorithms which compute them.

Venkatesh [91] introduced and claims to formally define the semantics of a variety of already existing forms of slice as well as introducing some of his own.

Like Weiser, his idea of a slice was not as a unique object. Slices are programs which preserve some projection of the semantics of the original program. Programs are all slices of themselves.

He defines a simple procedural language L with assignments, conditionals and loops (all statements being uniquely labelled). He defines a slice to be a set of labels, and defines a function '$syn(s, L)$' which constructs a legal program from a program and a subset of its labels in the obvious way. Although Venkatesh does not point it out, because of the properties of control dependence, the author believes that using Weiser's Algorithm, $L = syn(s, L)$. In other words the set of labels produced by Weiser's algorithm when applied to structured programs will be complete programs.

A set of labels L is a *static* backward end slice of p with respect to variable v if and only if for all states σ

$$\mathcal{M}[\![p]\!]\sigma(v) = \mathcal{M}[\![syn(p, L)]\!]\sigma(v)$$

A *Dynamic* Slice is defined in terms of an initial state σ_0. He defines L to be a dynamic slice with respect to program p and variable v if and only if

$$\mathcal{M}[\![p]\!]\sigma_0(v) = \mathcal{M}[\![syn(p, L)]\!]\sigma_0(v)$$

A static slice is therefore a special form of dynamic slice.

These definitions are somewhat inconsistent with the statement that 'program slices are only considered meaningful for terminating computations' as they clearly imply that the slice and the original program must agree even when the original program fails to terminate. Venkatesh probably meant:- A set of labels L is a *static* backward end slice of p with respect to variable v if and only if for all states σ

$$\mathcal{M}[\![p]\!]\sigma \neq \perp \implies \mathcal{M}[\![p]\!]\sigma(v) = \mathcal{M}[\![syn(p, L)]\!]\sigma(v)$$

Interestingly, he introduces the idea of a *Closure Slice*. A closure slice is just a set of labels that 'have an effect' on the variable(s) of interest. He states that

> 'In the case of closure slices, we are not interested in combining the labels to make a semantically correct subprogram.'

Algorithms which produce closure slices presumably, unlike Weiser's algorithm, have the property that $L \neq syn(L, s)$ and the behaviour of $syn(L, s)$ may not be the same as the behaviour of the original program (even with respect to the slice variable).

He defines closure slices in terms of *contamination*. A label l must be included in a closure slice with respect to variable v iff the effect of 'contaminating' the expression l percolates through to affect v. Although expressed completely differently, this is the same as label dependence introduced in this thesis.

Venkatesh gives a collection of formal definitions of different types of slice. These include the Dynamic Backward Closure Slice, the Dynamic Backward Executable Slice, Static Backward Closure Slice, Static Backward Executable Slice all of which, unlike Weiser's definition are functions.

He also introduces *forward slicing* which is where, given a variable v we are interested in all the expressions which are affected by the initial value of v. *Quasi–static slicing* is also introduced. This is simply where a program and its slice must agree not on a single state as in the dynamic case, nor on all states as in the static case, but on a prefix of the input.

In the final section of his paper Venkatesh introduces two algorithms one for dynamic and the other for static slicing. The latter he claims is equivalent to that used in [59] for inter–procedural slicing. His static slicing algorithm appears to produce the same slices as those of Weiser's algorithm. It is a reformulation of Weiser's algorithm using programs rather than control flow graphs. It has an accumulating parameter L for percolating control dependence information.

The main contribution of Venkatesh's work is that it introduces the idea that there are many different feasible semantic definitions of a slice. A major problem with the work, is that although his definitions are formal, they are not projections of the standard semantics, so have very little intuitive value in terms of program behaviour. The author believes that since slicing is about projecting properties of program behaviour on to a set of variables, a slice semantics is only useful if it is defined in terms of a program's standard semantics since that defines how programs behave when they are executed. It is not helpful to have to understand a whole new interpretation of a programming language in order to understand the meaning of a program slice.

2.5.10 Hausler's Work

Two years before Venkatesh, Hausler [52] states the same definition of a slice as Weiser. Namely that a slice S of P can be obtained from P by deleting zero or more statements and that if P halts on input i with values for the variables in the slicing criterion, then so does S with the same values for these variables. Hausler, like Venkatesh only considers end slicing. Although claiming to have given a denotational definition of a slice, he has really just written a slicing algorithm in a functional language. His algorithm is a dataflow algorithm (in the sense that it works at the level of abstraction of defined and referenced variables) and appears, like Venkatesh, to be another formulation of Weiser's Algorithm. The strength of Hausler's work lies in the fact that he expresses a slicing algorithm without explicitly mentioning a control flow graph. His algorithm works directly on programs. He does not explicitly use data and control dependence but they are, nevertheless encoded in his algorithm.

He uses two mutually recursive functions:

$$\delta : P \times \mathcal{P}V \to \mathcal{P}V$$

(which is his version of variable dependence) and

$$\alpha : P \times \mathcal{P}V \to P$$

which is the function that produces the slice.

δ is defined in terms of a function 'used' which is, in fact the referenced variables of an expression, so we shall call it '*ref*'. The rules for δ and α are now given:-

The abstract syntax of the language he considers is of the form:-

Γ ::= $x := E$ |
list(Γ) |
if B *then* Γ_0 *else* Γ_1 |
while B *do* Γ

For statement lists, the rules for δ are as follows:-

$$\delta(\text{nil}, V) = V$$
$$\delta(\text{append}(a, b)), V) = \delta(a, \delta(b, V))$$

and for single statements, the rules for δ are as follows:-

$$\delta(x := E, V) = \begin{cases} V - \{x\} \cup \mathit{ref}(E) & \text{if } x \in V \\ V & \text{otherwise} \end{cases}$$

$$\delta(\mathit{if}\ B\ \mathit{then}\ \Gamma_0\ \mathit{else}\ \Gamma_1, V) = \begin{cases} \mathit{ref}(B) \cup \delta(\Gamma_0, V) \cup \delta(\Gamma_1, V) & \text{if } \alpha(\Gamma_0) \neq \text{nil or } \alpha(\Gamma_1) \neq \text{nil} \\ V & \text{otherwise} \end{cases}$$

$$\delta(\mathit{while}\ B\ \mathit{do}\ \Gamma, V) = \bigcup_{n \in \mathbb{N}} \delta^n(\mathit{if}\ B\ \mathit{then}\ \Gamma\ \mathit{else}\ \text{nil}, V)$$

where $\delta^{n+1}(\Gamma, V) = \delta(\Gamma, \delta^n(\Gamma, V))$

and $\delta^0(\Gamma, V) = V$

For statement lists, the rules for α are as follows:-

$$\alpha(\text{nil}, V) = \mathit{nil}$$
$$\alpha(\text{append}(a, b)), V) = \text{append}(\alpha(a, \delta(b, V)), \alpha(b, V))$$

and for single statements, the rules for α are as follows:-

$$\alpha(x := E, V) = \begin{cases} x := E \text{ if } x \in V & \\ \text{nil} & \text{otherwise} \end{cases}$$

$$\alpha(\mathit{if}\ B\ \mathit{then}\ \Gamma_0\ \mathit{else}\ \Gamma_1, V) = \begin{cases} \mathit{if}\ B\ \mathit{then}\ \alpha(\Gamma_0, V)\ \mathit{else}\ \alpha(\Gamma_1, V) & \text{if } \alpha(\Gamma_0) \neq \text{nil or } \alpha(\Gamma_1) \neq \text{nil} \\ \text{nil} & \text{otherwise} \end{cases}$$

$$\alpha(\mathit{while}\ B\ \mathit{do}\ \Gamma, V) = \begin{cases} \mathit{while}\ B\ \mathit{do}\ \alpha(\ \Gamma, \bigcup_{n \in \mathbb{N}} \delta^n(\mathit{if}\ B\ \mathit{then}\ \Gamma\ \mathit{else}\ \text{nil}, V)) & \text{if } \alpha(\Gamma_0) \neq \text{nil} \\ \text{nil} & \text{otherwise} \end{cases}$$

This appears to be a reformulation of Weiser's Algorithm and is therefore not dataflow minimal. The parallel slicing algorithm [28] strongly resembles this. The accumulating parameter V corresponds exactly to the set of variables on each arc. Control flow is captured by the fact that a predicate is included if any of the statements within its body are included.

Importantly, Hausler proves that the above 'semantics' can be translated into an algorithm. The only part that was in question was the computability of

$$\bigcup_{n \in \mathbb{N}} \delta^n(\Gamma, V)$$

2.5.11 Unfolding

In his rule for δ applied to while loops he states:-

> 'Recall, semantically, the `while` statement is equivalent to `if then` statement composed with a `while` statement. i.e.
>
> $$\delta([\![\mathit{while}\ b\ \mathit{do}\ S]\!], V) = \delta([\![\mathit{if}\ b\ \mathit{then}\ S; \mathit{while}\ b\ \mathit{do}\ S]\!], V)$$
>
> In the δ definition, the `while` loop is just being unraveled, into one or more `if` statements composed together. δ^* accounts for zero or more iterations of the `while` loop.

...

It is not practical to iterate the loop an 'unknown' number of times. If the program slicer is to be used in an effective manner, the loop must be executed in order to compute δ^* for that loop, only a finite number of times. Fortunately, this is possible:

...

- First, it will be shown the number of times that a `while` loop has to be iterated in order to find the relevant variables can be computed.
- Secondly, this value can be computed primitive recursively.
- Thirdly, an easily computable upper bound exists on the number of necessary loop iterations. In fact, this upper bound can be computed from syntactic information alone, based on the type of statements in the body of the loop.'

Hausler proves that there is no problem with this as the construct is clearly monotonic and bounded above. He states:

> 'At some point in the computation, there will exist an n such that $\delta^{n-1}(\Gamma, V) = \delta^n(\Gamma, V)$ which implies that no new transitive effects were found. $\delta^{n+1}(\Gamma, V)$ does not have to be computed because nothing new will be added to δ^n.'

In the work introduced in this thesis, loops are unravelled[11] in a similar manner.

2.6 Dynamic Slicing

Dynamic Slices, introduced by Korel and Laski [71] have the potential to be much smaller than static ones since rather than having to agree in all states, a program and its dynamic slice need only behave the same in one particular given initial state. This initial state σ_0, say, becomes the third component of the slicing criterion in dynamic slicing.

Korel and Laski [71] were the first to introduce such a dynamic definition of a slice. A dynamic slice need only preserve the effect of the original program upon the slicing criterion

[11] We call it unfolding.

```
1    read(n);
2    s:=0;
3    p:=0;               p:=0;
     while (n>1)
     do begin
6      s:=s+n;
7      p:=p*n;
8      n:=n-1;
        end;
10   write(p);
```

Figure 2.14: Original Program and Dynamic Slice w.r.t. $(\{p\}, 10, <1>)$

when supplied with input x. The dynamic paradigm is ideally suited to bug-location, because a bug is typically detected as the result of the execution of a program with respect to some specific input, rather than by static consideration of the program's properties.

Consider the example in figure 2.14. The author of this program was hoping that it would output the product: $1 * \cdots * n$, where n is the value input. Suppose the original program has been executed and the value entered for the variable `n` was 1. The value printed at the end of the execution is incorrect — it is 0 when it should be 1. To locate the bug which causes this error a dynamic slice is constructed (see figure 2.14). The dynamic slice only identifies those statements which contribute to the value of the variable `p` when the input 1 is supplied to the program. Locating the bug (the faulty initialisation of `p`) in terms of the dynamic slice is thus easier than with either the original program or the corresponding static slice.

This is a rather extreme example of a dynamic slice, because the input causes the `while` loop to be ignored. However, dynamic slicing allows an improvement in precision in several ways. Clearly statements which remain unexecuted are not included in a dynamic slice. In addition, statements which are executed and create data and control dependencies may be removed from the slice should these dependencies be subsequently 'overwritten' during the execution.

In [5], Agrawal and Horgan use an ad–hoc approach to dynamic slicing that uses a program's program dependence graph. They give four algorithms for performing dynamic slicing.

The first two produce unnecessarily large slices. The third algorithm is the main one. It produces more accurate slices than the first two algorithms. The fourth algorithm is simply a more efficient version of the third and is not discussed here. Each algorithm involves first executing the program in state σ_0, and recording its *execution trace* as a finite sequence of nodes of the control flow graph that were visited.

Agrawal and Horgan's First Algorithm

Example

```
2     if x<0
      then
           begin
3            y:=f1(x);
4            z:=g1(x)
            end
      else
5          if x=0
           then
             begin
6              y:=f2(x);
7              z:=g2(x)
              end
            else
              begin
8              y:=f3(x);
9              z:=g3(x)
            end;
10    write(y);
11    write(z)
```

Figure 2.15: Example Program

Consider the program dependence graph of this program given in Figure 2.16(page 72) (We use dotted lines to represent control dependency and solid lines to represent data dependency).

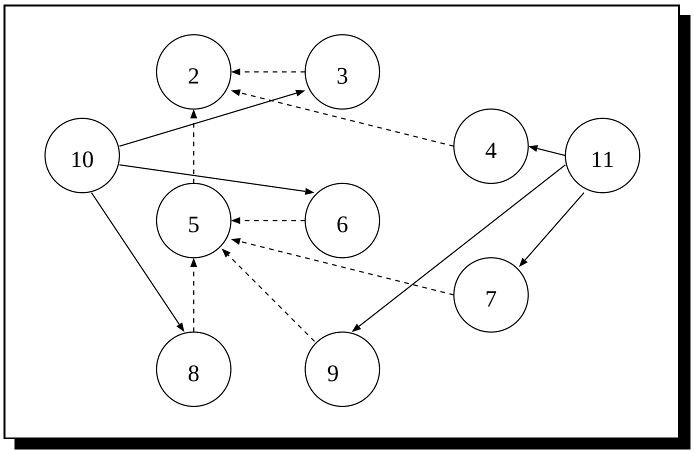


Figure 2.16: PDG

The standard program dependence graph approach to static slicing says that to compute the static slice with respect to variable y at line 10, we simply follow all the arrows starting at node 10. This gives $\{2, 3, 5, 6, 8, 10\}$.

Their first algorithm simply *intersects* the program dependence graph with the set of nodes that occur in the execution trace and then uses conventional static slicing on this reduced program dependence graph.

If we run the program in Figure 2.15(page 71) in an initial state with $x = -1$ we get the trace $< 2, 3, 4, 10, 11 >$.

Intersecting the program dependence graph in Figure 2.16(page 72) with $\{2, 3, 4, 10, 11\}$ gives the program dependence graph in Figure 2.17(page 72).

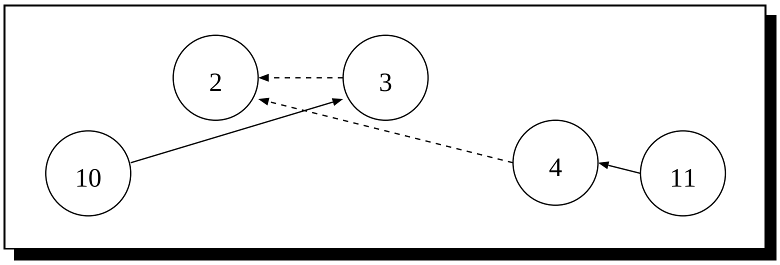


Figure 2.17: Reduced PDG

Slicing this program dependence graph with respect to y at node gives a correct and small dynamic slice $\{2, 3, 10\}$.

As pointed out in their paper this 'naive approach' does not lead to very accurate slices. Consider the program in Figure 2.18(page 73):

```
5      while i<n
       do  begin
6            z:=f1(z,y);
7            y:=f2(y);
8            i:=i+1
           end;
9       write(z)
```

Figure 2.18: Example

If we execute the program in Figure 2.18(page 73) in a state where $i = 0$ and $n = 1$ we get the execution trace $< 5, 6, 7, 8, 5, 9 >$.

The program dependence graph for this example is given in Figure 2.19(page 73)

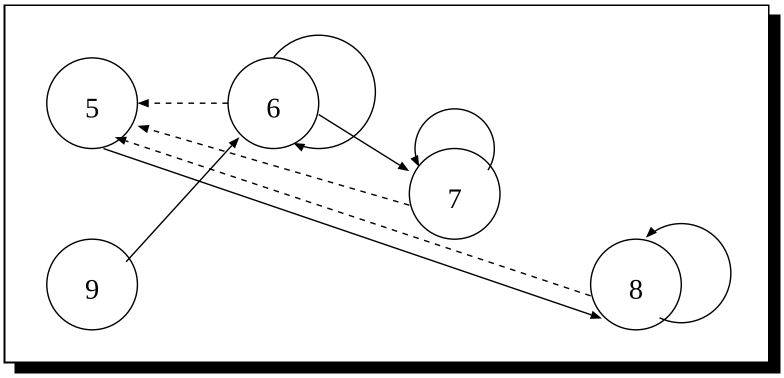


Figure 2.19: PDG

So if we intersect this program dependence graph with the trace $< 5, 6, 7, 8, 5, 9 >$ we get the whole program dependence graph. Static slicing this program dependence graph at node 9 with respect to z gives $\{5, 6, 7, 9\}$ so node 7 has been included unnecessarily. Since if the loop is executed only once, the assignment to y can have no effect on the final value of z.

Agrawal and Horgan's Second Algorithm

In Agrawal and Horgan's Second Algorithm, a reduced program dependence graph is constructed from the original program dependence graph and the trace. An arc from n_1 to n_2 in the program dependence graph is only made if n_2 occurs before n_1 in the trace.

Again, consider the program in Figure 2.18(page 73) with $i = 0$ and $n = 1$ which gives the trace

$$< 5, 6, 7, 8, 5, 9 > .$$

This approach yields the reduced program dependence graph in Figure 2.20(page 74).

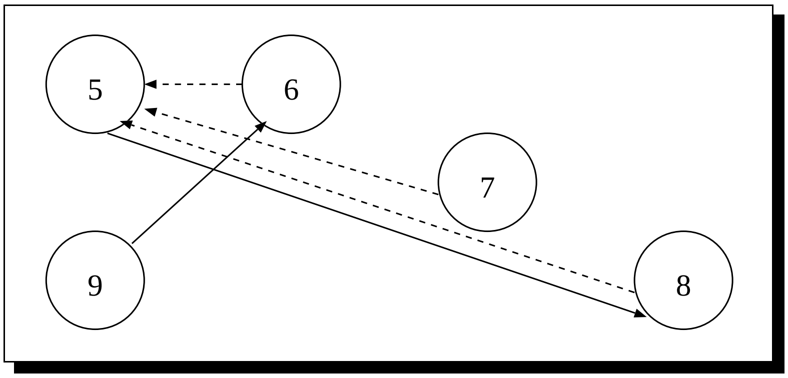


Figure 2.20: Reduced PDG

Slicing this program dependence graph starting at node 9 yields $\{5, 6, 8, 9\}$ and thus the offending node 7 is this time not included.

Agrawal and Horgan claim this approach also produces inaccurate slices. Consider the example program in Figure 2.21(page 74).

```
3         while i<n
          do
             begin
4               read(x);
5               if x<0
                then
6                   y:=f1(x)
                else
7                    y:=f2(x);
8               z:=f3(y);
9               write(z);
10              i:=i+1
             end
```

Figure 2.21: Example Program 3

This gives the PDG in Figure 2.22(page 75)

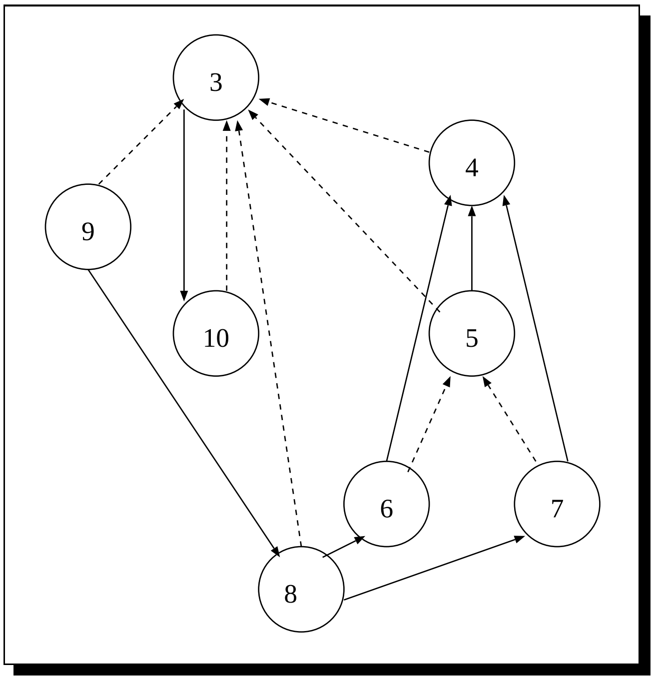


Figure 2.22: PDG

Consider the trace obtained from this program when the initial values are $i = 1$, $n = 3$ and the two values read in for x are -4 and 3. The trace produced is

$$< 3, 4, 5, 6, 8, 9, 10, 3, 4, 5, 7, 8, 9, 10, 3 > .$$

The approach described above leaves this program dependence graph in tact. Static slicing the program dependence graph with respect to variable z at node 9 gives rise to the whole program. Node 6 has unnecessarily been included as it has no effect on the final value of z in this case.

Agrawal and Horgan's Third Algorithm

Their third algorithm uses a **Dynamic Dependence Graph**.

To create the dynamic dependence graph, in effect, the program being sliced is first unfolded[12] as often as necessary, determined by the length of the trace. Each repeated node is treated as a new node The static slice of the resulting program dependence graph is then

[12] Agrawal and Horgan do not express it in terms of unfolding.

computed using the usual approach. A node is then deemed to be in the final slice if any of its instances are in the slice of the program dependence graph just described.

Consider again, the example program in Figure 2.21(page 74) for the case where `i`=1 and `n` =3 and the successive values input for `x` are -4,3, and -2. The trace of this is

$$< 3a, 4a, 5a, 6a, 8a, 9a, 10a, 3b, 4b, 5b, 7b, 8b, 9b, 10b, 3c, 4c, 5c, 6c, 8c, 9c, 10c, 3d >$$

Since the loop is executed three times the loop is unfolded three times to give the program in Figure 2.23(page 77).

```
3a      if i<n
        then
           begin
4a           read(x) ;
5a           if x<0
             then
6a               y:=f1(x)
             else
7a               y:=f2(x);
             z:=f3(y);
9a            write(z);
10a          i:=i+1;
3b             if i<n
               then
                 begin
4b                  read(x);
5b                  if x<0
                    then
6b                      y:=f1(x)
                    else
7b                     y:=f2(x);
8b                  z:=f3(y);
9b                  write(z);
10b                 i:=i+1;
3c                  if i<n
                    then
                      begin
4c                       read(x);
5c                       if x<0
                         then
6c                          y:=f1(x)
                         else
7c                          y:=f2(x);
8c                       z:=f3(y);
9c                       write(z);
10c                      i:=i+1
                      end
                 end
           end
```

Figure 2.23: Example Program 7

For each node in the element of the trace there will be a unique node in the dynamic dependence graph.

This gives the program dependence graph in Figure 2.24(page 78)

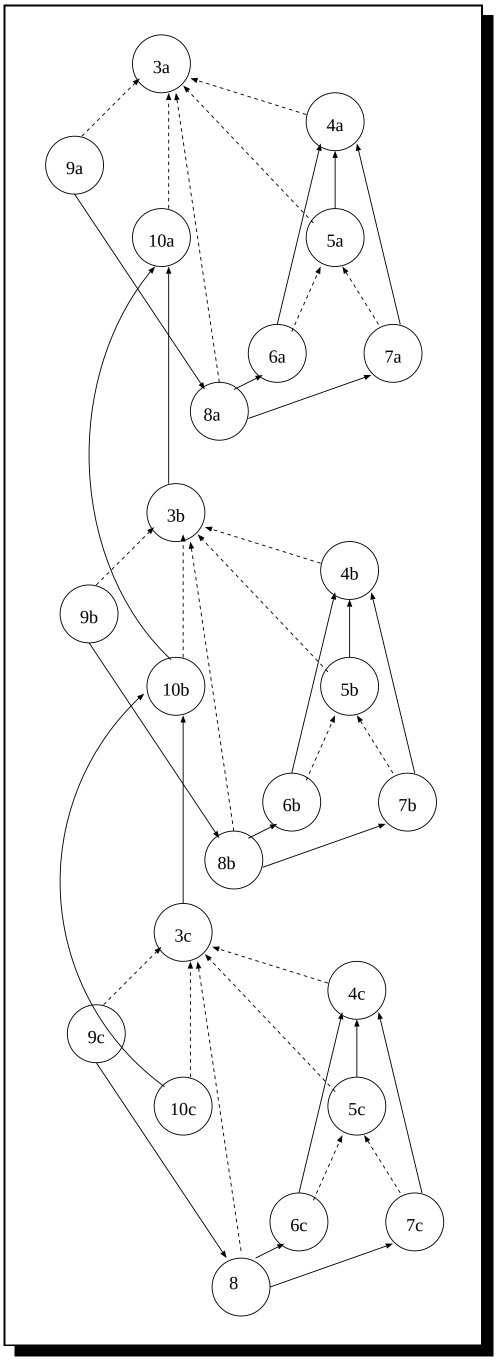


Figure 2.24: PDG

This yields the reduced program dependence graph in Figure 2.25(page 79)

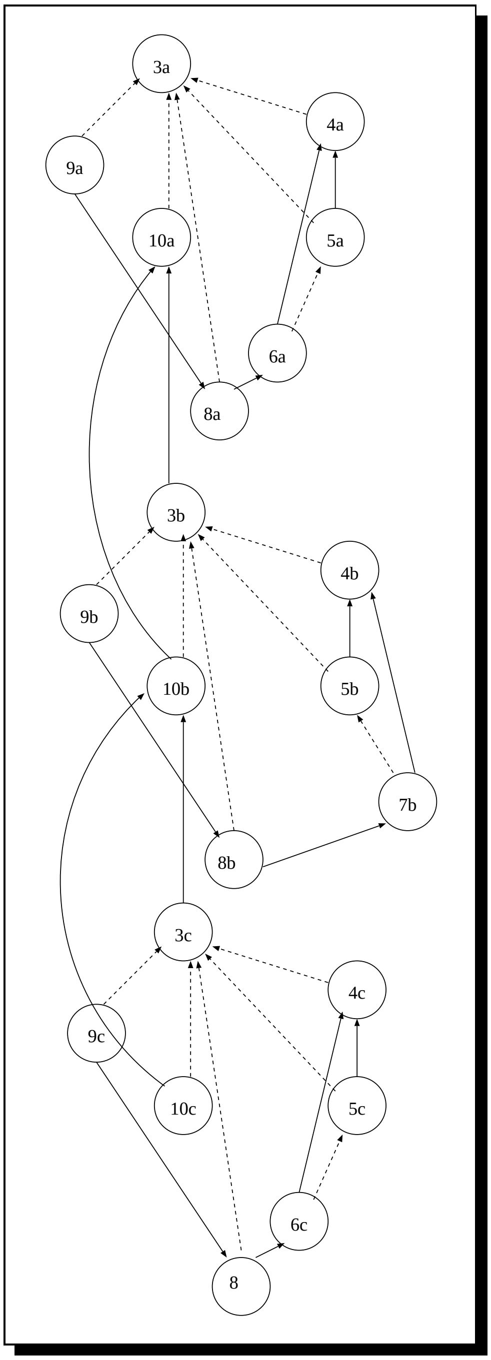


Figure 2.25: PDG

Agrawal and Horgan claim that to find the slice for a particular variable we simply find the node corresponding to the last definition of the variable and trace it back. This is only true for end slicing[13].

[13] There seems to be some confusion in their work between end and middle slicing. Strictly speaking we

In this case, we get the slicing on z give us node $9c$ which when traced back gives a slice $\{3, 4, 5, 6, 8, 9, 10\}$. As they rightly, point out, this does not include node 7. Using middle slicing however, we would have to trace back from *each* occurrence of the slicing criterion and so dynamic slicing at node 9 would be the whole program including node 7.

Their fourth algorithm is a more efficient approach to producing the same dynamic slice as produced by their third algorithm. It recognises like in [52] that programs need to be unfolded a fixed number of times (independent of the trace) to catch all necessary dependence information. A similar result is used in the main work of this thesis.

2.7 Symbolic Execution

The traditional execution model of program is based on the concept of *state*. Where a state is a mapping between variable names and *values*. The allowable values assigned to a variable v are defined by the *type* of v. The type of v may for example be `int` or `char`

In *symbolic execution* the situation is different. The values assigned to variables in a state are *symbolic expressions*. Each symbolic state is thus a representation of a whole set of traditional states.

An important example of symbolic execution is in proving the correctness of programs. In order to perform program proofs using Hoare Logic [54], a form of symbolic execution is undertaken.

Example

The weakest precondition rule for assignment is:-

$$\mathrm{wp}(x := E, R) = R[E/x]$$

This is interpreted as:

> In order to guarantee the truth of the predicate R after executing $x := E$ we must ensure that the predicate $R[E/x]$ is true before executing $x := E$.

cannot slice with respect to z using the given program dependence graph. A dummy statement referencing z the end of the program must first be included. Using statement 9 gives a *middle* slice.

The predicate $R[E/x]$ means R with all occurrences of the x replaced with the expression E. In other words, to compute $R[E/x]$ symbolically evaluate the predicate R in a symbolic state where the variable x has the symbolic value E.

Dannenberg and Ernst [29] explicitly use symbolic execution for program verification.

Symbolic Execution also occurs in the reduction of λ–expressions [24].

Besides proof of correctness, symbolic execution is used for assuring quality of software through testing. Symbolic execution was first used for program testing by King [69]. Here, symbolic execution is a form of static analysis, where the program is never actually executed. Instead, particular paths through the program are evaluated in detail. All the computations are performed symbolically, subject to constraints that may exist along the path because of the kind and number of conditionals that specify whether the path is executable. Howden [61] describes a symbolic testing and a symbolic execution system called DISSECT. The results of two classes of experiments in the use of symbolic execution are summarized. Several classes of program errors are defined and the reliability of symbolic testing in finding bugs is related to the classes of errors.

Huang [62] uses *symbolic traces* to increase error-detection capabilities of program tests and indicate the extent of their coverage. This instrumentation system generates traces automatically upon program execution.

Cimitile et al. [25] use symbolic execution in an approach to reverse engineering. With the help of theorem proving techniques they claim that they can recover the high level specification of functions from `C` programs. They are careful to point out that this process requires human interaction. They do not claim to have implemented their system.

Coen-Porisini et al. [27] use symbolic execution for software specialization.

Day [30] uses symbolic execution trees in in the functional language, Haskell [63], in a new automatic technique called symbolic functional evaluation (SFE) to evaluate semantic functions outside of a theorem proving environment. SFE produces the meaning of a specification.

These symbolic execution trees are defined as:-

```
data State f a =
       CondS a (State f a) (State f a) |
       Term (f a)
```

"A symbolic state captures a tree of possible execution paths that the machine could take."

Chapter 3

Dataflow Dependencies

3.1 Introduction

Dataflow analysis [92, 53], by definition, is the act of inferring properties about a program from its control flow graph alone. Dataflow analysis is, thus, fairly limited. We cannot, for example, tell by looking at a program's control flow graph when two expressions in the program are equal, nor can we use any form of expression simplification. All the information required to do such things has been 'abstracted away' in converting the program into a control flow graph.

Weiser's algorithm is an example of dataflow analysis. It takes the control flow graph g of a program p as input and outputs a set of nodes, N_g (a subset of the nodes of g). Since there is a one to one correspondence between the nodes of the control flow graph and the 'statements' of the corresponding program, this output uniquely determines which statements of p should be included in the slice of p. The slice, p', of p is the program *derived* from p and the set of nodes N_g output by Weiser's algorithm.

$$\begin{tikzcd} p \arrow[r, "\text{semantic relationship}"] \arrow[d, "\sim"'] & p' \\ g \arrow[r, "\text{Slice}"] & N_g(\mathit{set\ of\ nodes\ of\ g}) \arrow[u, "{f(N,p)}"'] \end{tikzcd}$$

Clearly, using dataflow analysis, all programs with the same control flow graph will be treated identically. Weiser noticed [92, 93] that his algorithm is not dataflow minimal. (See

Section 1.5, page 24 for some examples.) For some control flow graphs, g, there exists a set of nodes, N'_g which is a proper subset of N_g which has the property that for all programs p whose control flow graph is g, the required semantic relationship between p and $f(p, N'_g)$ is satisfied (where $f(p, N'_g)$ is the program derived from N'_g and p.).

A dataflow *minimal* algorithm A would be one for which no such smaller sets of nodes exist i.e. for every control flow graph g, the set of nodes A_g produced by A is such that for any proper subset N' of A_g, there will exist a program p whose control flow graph is g but the required semantic relationship between p and $f(p, N')$ is *not* satisfied.

Dataflow minimality is of interest, because if an algorithm for program analysis can be shown to be dataflow minimal, it is guaranteed to be the most precise algorithm achievable using dataflow analysis alone. Surprisingly, no work appears to have been done to investigate whether dataflow minimal slices are indeed achievable.

This chapter introduces eight dependence relations all defined in terms of the semantics of programs [88].

1. *VD* and *TVD*, both binary relations between the variables of a program.
2. *LD* and *TLD* both binary relations between the variables and labels of a program.

Programs with the same control flow graph are termed *dataflow equivalent*. Dataflow equivalence is formally defined and schemas [44] are used for representing classes of programs with the same control flow graph.

For each of the above four dependence relations on programs, equivalent ones are defined:

1. *DVD* and *DTVD*, both binary relations between the variables of a schema.
2. *DLD* and *DTLD* both binary relations between the variables and labels of a schema.

These 'dataflow dependencies', (two of which are a form of slicing) are defined in such a way such any algorithm for computing them *must* be dataflow minimal.

3.2 Minimal Slicing Algorithms

Definition 3.2.1 (Slice Preserving Algorithms)

Given an algorithm, A whose input objects are of type I and whose output objects are of type O. Given a *slice relation*: a binary relation R, between I and O, A is R–preserving if and only if for all i in I, $R(i, A(i))$ holds.

Definition 3.2.2 (Minimal Algorithms)

Furthermore, given an ordering $(\leq)$ on the elements of O, A is considered *R–minimal* if and only if for all o, $(o \lneqq A(i))$ implies that $R(i, o)$ does not hold.

These definitions are essentially, the 'equivalence relation' and the 'simplicity measure' described in [47].

3.2.1 Example: Statement Minimal Weiser Slices

A statement minimal end–slicing algorithm, with respect to variable x has input objects and output objects which are both programs. The ordering between programs is $p \sqsubseteq q$ if and only if p is a syntactic sub–component of q. The slice relation in this case, using Weiser's definition is $R_1(p, q)$ if and only if for all states when p terminates, q terminates with the same value for x.

3.2.2 Example: Dataflow Minimal Weiser Slicing

A dataflow minimal end–slicing algorithm, with respect to variable x has input objects which are control flow graphs and output objects which are sets of nodes. The ordering in this case is simply $\subseteq$. The slice relation, in this case, is given by:-
$R_2(g, N)$ if and only if for all p whose control flow graph is g, $R_1(p, q)$ where q is the program 'derived from' p and N. (R_1 is the slice relation defined in Section 3.2.1.)

Definition 3.2.3 (Dataflow Algorithms)

A dataflow algorithm is one whose inputs are control flow graphs, or some representation thereof.

Definition 3.2.4 (Dataflow Minimal Algorithms)

We define a dataflow minimal algorithm to be a dataflow algorithm that is minimal.

Applying Weiser's algorithm to the control flow graph, g, of $p_{1.2}$ in Figure 1.3(page 25) gives rise to the set of nodes $\{1, 2, 3, 4, 5\}$. It has been shown (Chapter 1, Section 1.5) that g and the set $\{1, 2, 4, 5\}$ satisfy the slice semantics with respect to end–slicing on variable x. The fact that $\{1, 2, 4, 5\} \subsetneqq \{1, 2, 3, 4, 5\}$ implies that Weiser's Algorithm is not dataflow minimal. (But it is dataflow.)

3.3 Discussion

It could be argued that Weiser's algorithm produces non–dataflow minimal slices only in 'pathological' cases. Nevertheless, it is not known how to predict when this situation will occur. When it does, the impact on the size of the slice, because of the transitivity of dependence, could be enormous. A single unnecessary extra node being included in a slice will cause the inclusion of all the nodes upon which this 'unnecessary' node depends.

'Traditional dependence'(see Section 2.2.4) is the transitive closure of the union of control and data dependence.

> What does it mean for node n to *depend* on node m (in the traditional sense i.e. $(D \cup C)^*$) in the control flow graph of p?

An attempt to semantically interpret the meaning of traditional dependence, could, for example, wrongly, but plausibly, postulate that node n_1 depends on node n_2 in a control flow graph g if and only if there exists a program p whose control flow graph is g in which the execution of the statement of p corresponding to node n_2 'has an effect' on the statement of p corresponding to node n_1. The phrase statement s_2 'has an effect' on the statement s_1 could be taken to mean: some execution of statement s_2 either affects the value of the expression in statement s_1 (data dependence) or affects whether a particular execution statement s_1 takes place at all (control dependence)[1].

As has been shown in the previous examples, there are cases when n_1 'depends on' n_2 with respect to control flow graph g but there are no programs in the equivalence class of q where the statement corresponding to n_2 has an effect on the statement corresponding to n_1. For example in program $p_{1.2}$ in Figure 1.2(page 24) node 4 depends on node 3 by transitivity, since node 2 is data dependent on node 3 and node 4 control dependent on node 2. There is no program in the same dataflow equivalence class of $p_{1.2}$ however, where the statement at node 3 has an effect on node 4.

What *can* be said semantically about traditional dependence is that if node n_1 is *not* dependent on node n_2 in a control flow graph g then in *all* programs whose control flow graph is g, the statement corresponding to n_1 will not be dependent on the statement corresponding

[1] This could be made more precise. But there no point as the statement is false!

to n_2. Clearly, traditional dependence is not the only relation between the nodes of a control flow graph that has this property, and what has been shown in the previous section, is that it is not the smallest. The dependence relation that relates *all* nodes of a control flow graph to each other would also produce a valid slicing algorithm, namely one that deletes nothing.

It is not sufficient to consider nodes on their own but on 'execution instances' of nodes: A node may get executed many times during the execution of a program, the ith execution of node n may be dependent on the jth execution of node m but not on the $j+1$th.

This chapter introduces program dependencies which do have a clear semantic interpretation. These dependencies are either between variables of a program (*VD* and *TVD*) or between variables of a program and nodes of its control flow graph, (*LD* and *TLD*).

For each of these dependencies d there will be a 'dataflow' version of d of the following general form:

> x *dataflow* d–depends on y in p means there exist a program q in the same dataflow equivalence class as p such that x *d–depends* on y in q.

For each dataflow dependence relation d, if an algorithm exists for computing it, it is guaranteed, by definition, to be dataflow minimal.

3.4 Assumptions about the Programming Language

Programs which are considered in this thesis are conventional imperative programs containing assignments, statement sequences conditionals and while loops as well as a *skip* statement which does nothing.

3.4.1 Syntax of Programs

$$
\begin{array}{rcl}
\Gamma & ::= & \mathit{skip} \mid \\
 & & \mathit{FAIL} \\
 & & x := E \mid \\
 & & \mathit{begin}\ \Gamma_1; \cdots ; \Gamma_n\ \mathit{end} \mid \\
 & & \mathit{if}\ B\ \mathit{then}\ \Gamma_0\ \mathit{else}\ \Gamma_1 \mid \\
 & & \mathit{while}\ B\ \mathit{do}\ \Gamma
\end{array}
$$

3.4.2 Types

There are no explicit type definitions in this programming language and it is further assumed that there are only two types of expression:

1. Boolean expressions that occur as the guards of ifs and whiles.

2. All expressions, which occur on the right hand side of assignments we assume to be of type integer.

3.4.3 The Variables Referenced by an Expression

The set of variables $ref(e)$ referenced by e is the set of variables that syntactically occur in the expression e. Clearly, by considering the expression $x - x$, it can be seen that if an expression 'depends upon' variable x then it references x but not necessarily vice–versa.

3.4.4 Assumptions about Expressions in Programs

Definition 3.4.1 (state)

A state is a finite mapping from variables to values.

For example the function

$$\sigma = \left\{ \begin{array}{c} x \mapsto 5 \\ y \mapsto 1 \\ z \mapsto 2 \end{array} \right\}$$

represents the state where variable x has the value 1, variable y has the value 1 and z the value 2. The reason that a state function is always finite is that a program can only assign values to a finite number of variables.

Definition 3.4.2 (The Domain of a State)

The *domain* of σ is the finite set of variables for which σ is defined.

For example, for the state σ above, $dom(\sigma) = \{x, y, z\}$.

Definition 3.4.3 (state projection)

Given any state σ, *proj* σ is the set of all states whose variables have the same value as those of σ and whose domains contain the domain of σ. (*proj* σ) defines an infinite set of states. We call such a set of states a *projection.*

For example,

$$\sigma_1 = \left\{ \begin{array}{c} w \mapsto 7 \\ x \mapsto 5 \\ y \mapsto 1 \\ z \mapsto 2 \end{array} \right\}$$

is such that $\sigma_1 \in$ *proj* σ.

Lemma 3.4.1 *Let e be an expression with $ref(e) = S$. Let σ be a state whose domain is S, then for all states σ' in $proj(\sigma)$,*

$$\mathcal{E}[\![e]\!]\sigma = \mathcal{E}[\![e]\!]\sigma'.$$

This result is intuitively obvious since if a variable is not mentioned in an expression, then it cannot have an effect on its value in any state.

We use σ in place of *proj* σ and $\mathcal{E}[\![e]\!]\sigma = z$ means for all σ' in *proj* σ $\mathcal{E}[\![e]\!]\sigma' = z$.

Assumption 3.4.1 (Richness of Expressions) *Given any finite set of states, $\sigma_1, \cdots, \sigma_n$ all with the same domain, S, and any set of values $v_1, \cdots, v_n$, there is an expression e with $ref(e) = S$, such that for all $i \in \{1, \cdots, n\}$,*

$$\mathcal{E}[\![e]\!]\sigma_i = v_i.$$

This assumption is used extensively in later proofs. It says that the expression notation of our programming language is sufficiently powerful such that given a finite set of states $\{\sigma_i\}$ and the same number of different values, $\{v_i\}$ we can pick an expression e, say, such that for each i, the expression, e, evaluated in state σ_i yields the value v_i.

3.4.5 Example

Let

$$\sigma_1 = \left\{ \begin{array}{c} x \mapsto 15 \\ y \mapsto 31 \\ z \mapsto 12 \end{array} \right\}$$

and

$$\sigma_2 = \left\{ \begin{array}{c} x \mapsto 5 \\ y \mapsto 1 \\ z \mapsto 2 \end{array} \right\}$$

Let us, for example, try to find an expression e which is such that $\mathcal{E}[\![e]\!]\sigma_1 = 79$ and $\mathcal{E}[\![e]\!]\sigma_2 = 81$.

It turns out that this assumption will be true provided that the arithmetic operators of our language include addition, subtraction, multiplication, and division. This result is a direct consequence of *Lagrange's Interpolation Formula* [16, 90]:

> Let F be a field, and let $a_0, \cdots, a_n$ be any distinct elements of F and let $c_0, \cdots, c_n$ be any given elements of F. There exists a polynomial $f(x)$ of degree $\leq n$ such that
>
> $$f(a_0) = a_0, \cdots, f(a_n) = c_n.$$

This result clearly generalises to where the a_i are states rather than simple values. Since all that is required is that each state, σ_i is first mapped each to a unique simple value before Lagrange's interpolation formula is applied.

For example, if the states are m–tuples $x_1, ..., x_m$ then

$$\prod_{i=1}^{n} \pi_i^{x_i}$$

where π_i is the ith prime number, is guaranteed to produce unique values for each distinct state.

Put another way: given any finite state to value function, there is an expression in our programming language that is *denoted by* that function.

Definition 3.4.4 (σ and σ' *differ only* on x)

Let σ and σ' be states with the same domain S and let x be a variable in S. Then σ and σ' *differ only* on x if and only if $\sigma\ x \neq \sigma'\ x$ and $\sigma\ z = \sigma'\ z$ for all $z \neq x$ in S.

Definition 3.4.5 (Affects)

Let e be an expression and let x be a variable. x *affects* e if and only if there exist two states σ and σ' differing only on x such that $\mathcal{E}[\![e]\!]\sigma \neq \mathcal{E}[\![e]\!]\sigma'$.

Lemma 3.4.2 *Let e be an expression. For all $x \notin ref(e)$, x does not affect E.*

Proof: obvious

Lemma 3.4.3 *Let T be a set of variables.*

There is an expression e with $ref(e) = T$ such that for all $x \in ref(e)$, x affects e.

Proof: Follows immediately from Assumption 3.4.1.

3.5 The Variable Dependencies: *VD* and *TVD*

In this section, two dependence relations on the variables of programs: *VD* and *TVD* are defined. Informally, they both have the property that x depends on y in p if the initial value of y *affects* the final value of x when executing program p. The difference between the two definitions arises from two different interpretations of the word 'affects'.

- In the case of *TVD*, a variable y is considered to affect x if and only if different values of y can cause p to **terminate** with different values for x.
- Using the more general, *VD*, on the other hand, y affects x either if
  - y affects x as just described or
  - if the initial value of y affects the termination of p.

Semantically, the difference between the two is that in DVD $\perp$ is considered a value whereas in $DTVD$, $\perp$ does not count.

3.5.1 Variable Dependence (VD)

Variable x is **variable dependent** on variable y in program p if and only if there exist two states, σ_1 and σ_2, differing only at y, such that either

- p terminates when started in state σ_1 and when started in state σ_2 with different final values for x.

  or

- either p terminates when started in state σ_1 or p terminates when started in state σ_2 but not both.

Formally:-

Definition 3.5.1 (Variable Dependence(VD))
Variable x is **variable dependent** on variable y in program p if and only if there exist two states, σ_1 and σ_2, differing only at y, such that

$$\mathcal{M}[\![p]\!]\sigma_1 x \neq \mathcal{M}[\![p]\!]\sigma_2 x.$$

We write $\underline{x \ VD \ y \text{ in } p}$.

In variable dependence, as opposed to terminating variable dependence(Section 3.5.3), $\perp$ is considered to be a value just like any other.

3.5.2 Examples of Variable Dependence

Example 1: `x` VD `x` in `skip`

Example 2: `x` VD `y` in `x:=y+1`

Example 3: `z` VD `z` in `x:=y+17`

Example 4: `x` VD `y` in `z:=y;x:=z`

Example 5: `x` VD `y` in `z:=y;a:=z;x:=a`

Example 6: `x` VD `y` in `if y=1 then x=1 else x=2`

Example 7: $\neg$(`x` VD `y` in `z:=y;x:=x+z-y`)

Example 8: `x` *VD* `x` in `while y<>0 do x:=x+1`

Example 9: `x` *VD* `y` in `while y<>0 do x:=x+1`

Example 10: `y` *VD* `y` in `while y<>0 do y:=y+1`

Lemma 3.5.1 *For all variables x and y, x VD y in skip $\iff$ $x = y$.*

Proof: Obvious.

Lemma 3.5.2 *For all terminating programs p, for all variables y not mentioned in p, y VD y in p.*

Proof: Obvious.

3.5.3 Terminating Variable Dependence (*TVD*)

Consider the program p_1 in Figure 3.1(page 93).

```
while y≠0
do x:=x-1
```

Figure 3.1: Program p_1

Clearly, the final value of the variable x is dependent on the initial value of y (and the initial value of x). There are however no initial states differing only at y such that p_1 **terminates** with different final values for x. In this sense the final value of x is **not** dependent on the initial value of y. This observation leads to a new version of variable dependence of programs which we call **Terminating Variable Dependence**.

Variable x is **terminating variable dependent** on variable y in program p if and only if there exist two states, σ_1 and σ_2, differing only at y, such that p terminates when started in states σ_1 and when started in state σ_2 with different final values for x. Formally:-

Definition 3.5.2 (*TVD*)

Variable x is **terminating variable dependent** upon y in program p if and only if there exist two states σ and σ' differing only at variable y, such that

$$\perp \neq \mathcal{M}[\![p]\!]\sigma\ x \neq \mathcal{M}[\![p]\!]\sigma'\ x \neq \perp$$

We write $\underline{x\ TVD\ y \text{ in } p}$.

For terminating variable dependence, therefore, $\perp$ 'does not count' as a value.

3.5.4 Examples of Terminating Variable Dependence

Example 1: `x` TVD `x` in `skip`

Example 2: `x` TVD `y` in `x:=y+1`

Example 3: `x` TVD `y` in `x:=y+17`

Example 4: `x` TVD `y` in `z:=y;x:=z`

Example 5: `x` TVD `y` in `z:=y;a:=z;x:=a`

Example 6: `x` TVD `y` in `if y=1 then x=1 else x=2`

Example 7: $\neg$(`x` TVD `y` in `z:=y;x:=x+z-y`)

Example 8: `x` TVD `x` in `while y<>0 do x:=x+1`

Example 9: `x` TVD `y` in `while y<>0 do x:=x+1`

Example 10: $\neg$(`y` TVD `y` in `while y<>0 do y:=y+1`)

If the examples above, if VD and TVD are compared, it can be seen that a difference occurs only in `while y<>0 do y:=y+1`. Variable `y` variable depends on `y` since the initial value of `y` can determine whether the program terminates or not. In terminating variable dependence we are only interested in terminating programs and in these variable `y` always ends up with the value zero. Using TVD therefore, `y` is dependent on no variables at all.

Lemma 3.5.3

$$x\ TVD\ y \text{ in } p \implies x\ VD\ y \text{ in } p$$

Proof: trivial.

The converse is clearly not true.

3.6 The Undecidability of *VD* and *TVD*

Lemma 3.6.1 *VD is undecidable.*

Proof:

If an algorithm could be written to decide whether x *VD* y in program p then we could use it to solve the halting problem [78] as follows:- In order to decide whether program p halts, construct the program q given by:

```
p;
y:=z;
```

Where `y` and `z` are variables not occurring in p. Then p halts if and only if `y` *VD* `z` in q. Similarly,

Lemma 3.6.2 *TVD is undecidable.*

Proof: Program q also has the property that p halts if and only if `y` *TVD* `z` in q.

3.7 Dataflow Dependence

In this section, since dataflow analysis is our concern, the definitions of variable dependence and terminating variable dependence are recast in terms of control flow graphs rather than programs.

Clearly, if we know only the variables referenced by each expression in a program we cannot know the variable dependence of the original program since crucial information has been abstracted away.

```
x:=y;
x:=x-y
```

Figure 3.2: Program p_2

In program p_2 in Figure 3.2(page 95) the final value of `x` is not dependent on the initial value of `y`. In program p_3 in Figure 3.3(page 96), however, the final value of `x` clearly *is* dependent

```
x:=y
x:=x-2*y
```

Figure 3.3: Program p_3

on the initial value of `y`. This is an example of two programs with the same control flow graph but with different variable dependence.

As described in Section 3.1, the programs p_1 and p_2 in Figure 3.2(page 95) and Figure 3.3(page 96) although different, can be considered to be in the same equivalence class, since they have same control flow graph, i.e. q can be obtained from p by replacing expressions by other expressions which reference the same sets of variables. The programs, p_1 and p_2 are an example of dataflow equivalent programs. We write $p_1 \sim p_2$ (see Definition 3.7.1(page 97)).

Since, as has been just shown, it is not possible to write an algorithm for checking variable dependence, 'cruder' problems will now be investigated. If we group programs together into equivalence classes, maybe there exist algorithms which can answer the question:

> Is there any program q 'equivalent' to p such that x depends on y in q?

If each program is put in its own equivalence class then, by Lemma 3.6.1(page 95) and Lemma 3.6.2(page 95), the problem is undecidable. Clearly, if we put all programs into the same equivalence class, then the problem is trivial. A main concern of this thesis is to investigate whether these problems are solvable when the equivalence is *dataflow equivalence* and if so, to produce algorithms for solving them.

3.7.1 Dataflow Equivalence

In this section, the notion of dataflow equivalence is formalised. Rules for dataflow equivalence corresponding to each syntactic category of program are given. Programs are dataflow equivalent if they have the same control flow graph. Since we are just considering structured programs[2], it can be said that dataflow equivalent programs have identical structure up to expressions and 'corresponding' expressions reference the same sets of variables. Also the

[2] gotos are not allowed.

```
while x < y
do if x < y
  then x:=x + y
  else y:=y+1
```

```
while 3*x+2*y=17
do if x>y-y
  then x:=x+1+y
  else y:=y+1
```

```
while x+x+y=y+1
do if x=y+x-y
  then x:=x-x+y-y
  else y:=y
```

Figure 3.4: Three Dataflow Equivalent Programs

variables occurring on the left hand sides of corresponding assignment statements must be the same.

Definition 3.7.1 (Dataflow Equivalence)

$$skip \sim skip$$

$$\frac{ref(e) = ref(e')}{v := e \sim v := e'}$$

$$\frac{s_1 \sim s_1', \cdots, s_n \sim s_n'}{begin\ s_1; \cdots; s_n\ end \sim begin\ s_1'; \cdots; s_n' end}$$

$$\frac{ref(e) = ref(e') \quad s_1 \sim s_1' \quad s_2 \sim s_2'}{if\ e\ then\ s_1\ else\ s_2 \sim if\ e'\ then\ s_1'\ else\ s_2'}$$

$$\frac{ref(e) = ref(e') \quad s \sim s'}{while\ e\ do\ s \sim while\ e'\ do\ s'}$$

3.7.2 Example of Dataflow Equivalence

In Figure 3.4(page 97) three dataflow equivalent programs are given. They all have the control flow graph given in Figure 3.5(page 98).

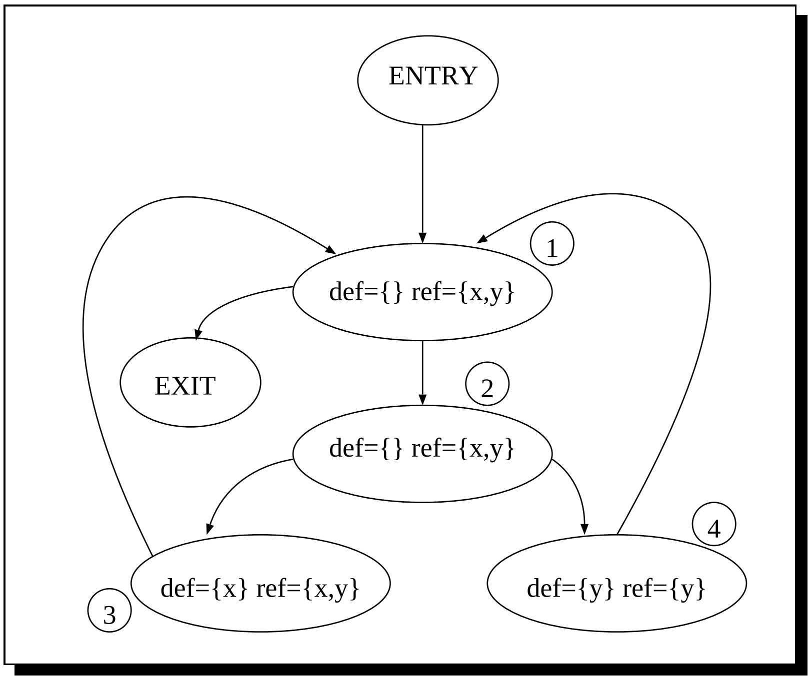


Figure 3.5: Control Flow Graph

3.8 The Dataflow Variable Dependencies: DVD and $DTVD$

The two versions of variable dependence, VD and TVD(Definition 3.5.1(page 92) and Definition 3.5.2(page 93)) each have dataflow versions DVD and $DTVD$ which are now defined.

3.8.1 Dataflow Variable Dependence(DVD)

Variable x is **dataflow variable dependent** upon y in program p if and only if there exists a program q with the same control flow graph as p such that x VD y in q. Formally:-

Definition 3.8.1 (DVD)
Variable x is **dataflow variable dependent** upon y in program p if and only if there exists $q \sim p$ such that x VD y in q.
We write $\underline{x\ DVD\ y \text{ in } p}$.

3.8.2 Examples of Dataflow Variable Dependence

Example 1: `x` DVD `x` in `skip`

Example 2: `x` *DVD* `y` in `x:=y+1`

Example 3: `z` *DVD* `z` in `z:=z-z`

Example 4: `x` *DVD* `y` in `z:=y;x:=z`

Example 5: `x` *DVD* `y` in `z:=y;a:=z;x:=a`

Example 6: `x` *DVD* `y` in `if y=1 then x=1 else x=1`

Example 7: `x` *DVD* `y` in `z:=y;x:=x+z-y`

Example 8: `x` *DVD* `x` in `while y<>0 do x:=x+1`

Example 9: `x` *DVD* `y` in `while y<>0 do x:=x+1`

Example 10: `y` *DVD* `y` in `while y<>0 do y:=y+1`

3.8.3 Dataflow Terminating Variable Dependence($DTVD$)

Variable x is **dataflow terminating variable dependent** upon y in program p if and only if there exists a program q with the same control flow graph as p such that x TVD y in q. Formally:-

Definition 3.8.2 ($DTVD$)
Variable x is **dataflow terminating variable dependent** upon y in program p if and only if there exists $q{\sim}p$ such that x TVD y in q.
We write $\underline{x\ DTVD\ y \text{ in } p}$.

Examples of Dataflow Terminating Variable Dependence are given in the next section.

3.8.4 A Taxonomy of Variable Dependence

So far, four dependencies have been introduced:

- VD together with its dataflow counterpart, DVD
  and
- TVD together with its dataflow counterpart, $DTVD$.

These four different variable dependencies can be categorised as follows:-

$$
\begin{array}{lcl}
VD & = & \text{(non–terminating,normal)} \\
TVD & = & \text{(terminating,normal)} \\
DVD & = & \text{(non–terminating,dataflow)} \\
DTVD & = & \text{(terminating,dataflow)}
\end{array}
$$

The definition of each can be tabulated as follows:

x VD y in p	$\exists$ σ and σ' differing only at y such that $\mathcal{M}[\![p]\!]\sigma x \neq \mathcal{M}[\![p]\!]\sigma' x$
x TVD y in p	$\exists$ σ and σ' differing only at y such that $\bot \neq \mathcal{M}[\![p]\!]\sigma x \neq \mathcal{M}[\![p]\!]\sigma' x \neq \bot$
x DVD y in p	$\exists q \sim p$ such that x VD y in q
x $DTVD$ y in p	$\exists q \sim p$ such that x TVD y in q

Figure 3.6: The Four Variations of Variable Dependence

For each pair of definitions, we give an example program where the respective dependencies are different.

1. VD $\neq$ DVD.
   In program p_2, Figure 3.2(page 95), $\neg(x$ VD $y)$ but x DVD y.

2. VD $\neq$ TVD.

   Consider program p_{10} in Figure 3.7(page 101). If we start in a state where y is negative, then the value for x in the final state is $\bot$. However if we start in a state where y is non–negative, then the final value of x is zero, so x VD y. But for all states where the program terminates the final value of x is zero, independent of the value of y, so $\neg(x$ TVD $y)$.

```
while y≠0
do y:=y-1;
x:=y
```

Figure 3.7: program p_{10}

3. TVD $\neq$ DTVD.

   Again, consider program p_{10} in Figure 3.7(page 101). Consider also the dataflow equivalent program p_9 in Figure 3.8(page 101).

```
while y>0
do y:=y-1;
x:=y
```

Figure 3.8: program p_9

   In program p_9, (x TVD y), so by definition since p_9 and p_{10} are equivalent, we have (x DTVD y) in p_{10}. As we showed earlier, in p_{10}, $\neg$(x TVD y).

4. VD $\neq$ DTVD.

   Consider again program p_2 in Figure 3.2(page 95). $\neg$(x VD y) but x DTVD y.

5. TVD $\neq$ DVD.

   Consider again program p_2 in Figure 3.2(page 95). $\neg$(x VD y) but x DVD y.

6. DVD $\neq$ DTVD.

   Consider program p_{12} in Figure 3.9(page 102). $\neg$(x DTVD y) since for all programs q dataflow equivalent to p_{12}, if q terminates, the final value of x will not depend on the value of y in the initial state. On the other hand x DVD y since the initial value of y affects termination of p_{12}.

```
while y>0
do y:=y-1;
x:=7
```

Figure 3.9: program p_{12}

3.9 The Label Dependencies: *LD* and *TLD*

A form of dependence more closely related to slicing than variable dependence is now introduced. First, programs are labelled[3] with names corresponding to the node identifiers in their control flow graph. Informally, a variable x is label dependent on label l if and only if changing the expression at label l to another one that references the same set of variables can affect the final value of x. This is similar to the idea of *contamination* of expressions [91]. The 'slice' on x produced by label dependence will be the set of all the labels whose expressions can affect x in this way. It is a closure slice [91] as it is a collection of labels that do not necessarily make up a complete program. As in the case of variable dependence, there are two variants: *terminating* and *non-terminating*. Again, these arise from the two different interpretation of the word 'affects'. (See Section 3.5, page 91 for an explanation of this issue.)

3.9.1 Label Dependence (*LD*)

Definition 3.9.1 (*LD*)

Variable x is **label dependent** on label l in p if and only if there exists a program p' dataflow equivalent to p, differing from p' only at label l, and a state, σ, such that

$$\mathcal{M}[\![p]\!]\sigma x \neq \mathcal{M}[\![p']\!]\sigma x.$$

We write $\underline{x\ LD\ l \text{ in } p}$.

For example consider the program in Figure 3.10(page 104). The program, `1: x:=y+1` is dataflow equivalent to it and differs only at the expression labelled 1. Clearly the final values of x will be different for the two programs starting in any state where y is defined.

[3] We do not think of these labels as part of the language, but as comments.

3.9.2 Terminating Label Dependence (*TLD*)

Definition 3.9.2 (*TLD*)

Variable x is **terminating label dependent** on label l in p if and only if there exists a program p' dataflow equivalent to p, differing from p' only at label l, and a state, σ, such that

$$\perp \neq \mathcal{M}[\![p]\!]\sigma x \neq \mathcal{M}[\![p']\!]\sigma x \perp \neq \perp.$$

We write $\underline{x\ \mathit{TLD}\ l \text{ in } p}$.

As in the case of variable dependence, we have:-

Lemma 3.9.1

$$x\ \mathit{TLD}\ L \text{ in } p \implies x\ \mathit{LD}\ L \text{ in } p.$$

Proof: trivial.

In Figure 3.18(page 106) and Figure 3.19(page 106) we have examples of two programs with the same *LD* but different *TLD*. The only way the program in Figure 3.19(page 106) can terminate is with $x = 0$. The expression at label 2 cannot affect this. It can only affect termination conditions of the program.

The dataflow variants of these dependencies are produced in exactly the same way as the dataflow variants of variable dependence. Variable x is **dataflow label dependent** on label l in p means there exist two programs q and q' both dataflow equivalent to p which differ only at l such that there exists a state σ where q and q' 'behave differently' with respect to x when started in state σ.

Definition 3.9.3 (*DLD*)

Variable x is **dataflow label dependent** upon l in program p if and only if there exists $q{\sim}p$ such that $x\ \mathit{LD}\ l$ in q.

We write $\underline{x\ \mathit{DLD}\ l \text{ in } p}$.

Definition 3.9.4 (*DTLD*)

Variable x is **dataflow terminating variable dependent** upon l in program p if and only if there exists $q{\sim}p$ such that $x\ \mathit{TLD}\ l$ in q.

We write $\underline{x\ \mathit{DTLD}\ l \text{ in } p}$.

```
1:  x:=y
```

Figure 3.10: $x\,VD\{y\}$ and $x\,LD\{1\}$

```
1:  x:=5
```

Figure 3.11: $x\,VD\{\}$ and $x\,LD\{1\}$

3.9.3 Examples

Figures 3.10 to 3.10 show examples comparing the various dependencies. The interesting examples involve loops. Notice the subtle difference between Figure 3.18(page 106) and Figure 3.19(page 106). In figure 3.18, if we start in a state when y is negative then the program will terminate with x having the initial value of y. In figure 3.19 however, in all states when the program terminates x will have the value 0. The initial value of y only affects termination and not the final value of x according to definition 3.5.1. In figure 3.19, therefore the *VD* and the *TVD* are different since this value is independent of the initial value of y. Similarly the assignment `y:=y-1` has no effect on the final value of x. Even if this had said `y:=y+79`, the only final value of x could be zero. The same thing happens in the program in Figure 3.20(page 107). Label 6 has no effect on the final value of x in terminating programs but it can affect the termination of the program. It therefore occurs in *LD* but not in *TLD*.

```
1:  x:=z
```

Figure 3.12: $x\,VD\{z\}$ and $x\,LD\{1\}$

```
1:  z:=y;
2:  x:=y+4
```

Figure 3.13: $x\,VD\{y\}$ and $x\,LD\{2\}$

```
1:  if y=2
2:  then x:=25
3:  else x:=25
```

Figure 3.14: $x\,VD\{\}$ and $x\,LD\{2,3\}$

```
1:  if y=2
2:  then x:=17
```

Figure 3.15: $x\,VD\{x,y\}$ and $x\,LD\{1,2\}$

```
1:  z:=y;
2:  if z=2
3:  then x:=17
```

Figure 3.16: $x\,VD\{x,y\}$ and $x\,LD\{1,2,3\}$

```
0:  x=0;
1:  while y>0
    do
    begin
2:     x:=x+1;
3:     y:=y-1
    end
```

Figure 3.17: $x\,VD\{y\}$ and $x\,TVD\{y\}$ and $x\,LD\{0,1,2,3\}$ and $x\,TLD\{0,1,2,3\}$

```
1:  while y>0
2:  do y:=y-1;
3:  x:=y
```

Figure 3.18: $x\,VD\{y\}$ and $x\,TVD\{y\}$ and $x\,LD\{1,2,3\}$ and $x\,TLD\{1,2,3\}$

```
1:  while y <> 0
2:  do y:=y-1;
3:  x:=y
```

Figure 3.19: $x\,VD\{y\}$ and $x\,TVD\{\}$ and $x\,LD\{1,2,3\}$ and $x\,TLD\{1,3\}$

```
2: while i<3
   do
   begin
3:  if c=2
    then
     begin
4:     c:=y;
5:     x:=25
     end
6:  i:=i+1
   end
```

Figure 3.20: $x\,VD\{x,c,i\}$ and$x\,TVD\{x,c,i\}$ $x\,LD\{2,3,5,6\}$ and $x\,TLD\{2,3,5\}$

3.9.4 A Taxonomy of Label Dependence

So far, four dependencies have been introduced:

- *LD* together with its dataflow counterpart, *DLD*

  and

- *TLD* together with its dataflow counterpart, *DTLD*.

These four different label dependencies can be categorised as follows:-

$$
\begin{aligned}
LD &= \text{(non–terminating,normal)} \\
TLD &= \text{(terminating,normal)} \\
DLD &= \text{(non–terminating,dataflow)} \\
DTLD &= \text{(terminating,dataflow)}
\end{aligned}
$$

x *LD* l in p	$\exists\ p'{\sim}p$ differing from p only at l, and a state σ such that $\mathcal{M}[\![p]\!]\sigma x \neq \mathcal{M}[\![p']\!]\sigma x$
x *TLD* l in p	$\exists\ p'{\sim}p$ differing from p only at l, and a state σ such that $\perp \neq \mathcal{M}[\![p]\!]\sigma x \neq \mathcal{M}[\![p']\!]\sigma x \neq \perp$
x *DLD* l in p	$\exists q{\sim}p$ such that x *LD* l in q
x *DTLD* l in p	$\exists q{\sim}p$ such that x *TLD* l in q

Figure 3.21: The Four Variations of Label Dependence

3.10 The Undecidability of *LD* and *TLD*

Lemma 3.10.1 *LD is undecidable.*

Proof: As in the case of variable dependence, if an algorithm could be written to decide whether x *LD* l in program p then we could use it to solve the halting problem [78] as follows:- In order to decide whether program p halts, construct the program q given by:

```
p;
l:y:=5;
```

Then p halts if and only if `y` *LD* `l` in q. Similarly,

Lemma 3.10.2 *TLD is undecidable.*

Proof: Program q also has the property that p halts if and only if `y` *TLD* `l` in q.

3.11 Schemas

Before our new dependencies are further investigated, notation for representing dataflow equivalence classes of programs is introduced[4]. The only difference between a program and a

[4] This notation is very similar to that used in *schemes* [44].

schema is that where the former has expressions, the latter has labelled sets of variables. We call these labelled sets of variables, *Symbolic Expressions.*

3.11.1 Syntax of Schemas

$$
\begin{array}{rcl}
\Gamma & ::= & \textit{skip} \mid \\
 & & x := L(\mathcal{P}(V)) \mid \\
 & & \textit{begin } \Gamma_1; \cdots ; \Gamma_n \textit{ end} \mid \\
 & & \textit{if } L(\mathcal{P}(V)) \textit{ then } \Gamma_0 \textit{ else } \Gamma_1 \mid \\
 & & \textit{while } L(\mathcal{P}(V)) \textit{ do } \Gamma \mid \\
 & & \textit{FAIL}
\end{array}
$$

A schema is a notation that can be used for representing *Dataflow Equivalence classes* of programs. The use of *FAIL* is explained later. For example, the simple schema, $s_{3.22}$ in Figure 3.22(page 109) represents the dataflow equivalence class containing the program in Figure 3.20(page 107).

```
while f1(i)
do
begin
  if f2(c)
  then
    begin
      c := f3(y);
      x := f4()
    end;
   i := f5(i)
end
```

Figure 3.22: $s_{3.22}$, the Schema corresponding to the program in Figure 3.20(page 107).

3.11.2 Uniqueness of Labels

In the theory of schemes [44] the same symbolic expression can occur at different places. Different occurrences of the same term correspond to programs with the same expression occurring in different places. Although, in practice, the same expression can occur in different places in a program, taking advantage of it is not allowable in dataflow analysis, where all that is known about each expression is its set of referenced variables. The fact that two expressions are the same is invisible to an agent that performs dataflow analysis. In performing dataflow analysis, therefore, we imagine all programs have been first translated into schemas where there is no repetition of labels. As will be seen in Chapter 7, as a result of unfolding schemas, however, repetitions of the same labels do need to be considered.

The theory introduced in this thesis *does* allow schemas with repetitions of labels. Unlike schemas with no repetition of labels, schemas with repeated labels do not represent dataflow equivalence classes of programs since allowing repetition means that the classes of programs represented by schemas are not disjoint: for example the program `x:=x+y;x:=x+y` is both represented by the schema `x:=f(x,y);x=f(x,y)` which represents all program consisting of two assignments to variable x of the *same* expression referencing x and y and also by the schema `x:=f(x,y);x=g(x,y)` which represents all programs consisting of two assignments to variable x of *any* expression referencing x and y. (Here the expression may or may not be the same in the two assignments).

Definition 3.11.1 (The class of Programs represented by a Schema)
We write $[s]$ for *the class of programs represented by* s.

Lemma 3.11.1 *If each label of a schema* s *occurs only once in* s*, then* $[s]$ *is a dataflow equivalence class.*

Proof: Follows from definition of dataflow equivalence, Definition 3.7.1(page 97).

3.11.3 Interpretations

Every expression in a conventional programming language is a syntactic representation of an *expression function*, $\mathcal{E}[\![E]\!]$ which is a function from states to values [88]. In a schema, a symbolic expression is a representation of the set of all possible *expression functions* that could occur in its place in any dataflow equivalent program. The label represents the name of the function and the set of variables those that are referenced in the corresponding expression in the program.

Given a schema s, borrowing the terminology of Greibach [44], a program p in the class represented by s is called an **interpretation** of s. We can think of a program, therefore as a schema s together with a function f, say, from the labels of s to the expressions of p, such that for all symbolic expressions $l(V)$ of s, $ref\ f(l) = V$.

We will sometimes abuse notation and associate a program p in $[s]$ with such a function and write '$p(l)$' for the expression of p that corresponds to the symbolic expression labelled l in s. For example, the function which gives the correspondence between schema $s_{3.22}$ and the program in Figure 3.20(page 107) is given by:-

$p(f1) = (i < 3)$

$p(f2) = (c = 2)$

$p(f3) = y$

$p(f4) = 25$

$p(f5) = i + 1$

As in the case of theory of Schemes [44], great care has to be taken to properly define the domain of interpretation, that is the set of allowable values for expressions. This choice as in the case of the theory of schemes can greatly affect fundamental properties of schemas. For our theory and our algorithms to be correct it is important that the domains are infinite; without loss of generality, we can therefore assume that the expressions on the right hand of assignments are of type integer.

Since this thesis is about dataflow dependence, from now on, schemas rather than programs will be used as the objects to which dependence relations are applied.

3.12 Redefining our Dataflow Dependencies in terms of Schemas

In this section our four different dataflow dependence relations are recast in terms of schemas. The four dataflow dependecies are:

$$\begin{array}{lcl} xDVDy \text{ in } s & \iff & \exists p \in [s] \text{ such that } xVDy \text{ in } p. \\ xDTVDy \text{ in } s & \iff & \exists p \in [s] \text{ such that } xTVDy \text{ in } p. \\ xDLDy \text{ in } s & \iff & \exists p \in [s] \text{ such that } xLDy \text{ in } p. \\ xDTLDy \text{ in } s & \iff & \exists p \in [s] \text{ such that } xTLDy \text{ in } p. \end{array}$$

3.13 A Comparison of Dataflow Label Dependence with Slicing

In this section it is informally justified that *DTLD* is the dataflow minimal version of Venkatesh's static backward closure slice [91] and it also claimed that *DLD* is the dataflow minimal version of slicing that preserves the projection, onto the variable of interest, of *standard* (rather than lazy[5]) semantics. (Kamkar [66] introduces this slice semantics in her thesis and so we refer to slices satisfying this slice relation as 'Kamkar Slices').

It is claimed therefore that the slices produced by *DLD* are always executable and always behave identically with respect to the variable of interest. Not only is the value of the variable of interest preserved by the slice, but also the original program and the slice will always agree in terms of termination.

3.13.1 The Slices produced by *DTLD*

Venkatesh [91] states:

> 'Intuitively, a statement belongs to a slice if and only if the statement's computation contributes to the value of the variable of interest. Suppose we could selectively *contaminate* the computation of a statement in such a way that the contamination propagated to all variables that depended on that statement. Then we could use this as a test to check whether a statement is necessary to compute the value of a specified variable. On the other hand, a statement does not belong to a slice if its contamination does not affect the variable of interest. Contamination of the computation of such a statement must not contaminate the specified variable'

[5] as in the case of Weiser's algorithm [92] and the PDG approach [82]

The idea of selective contamination corresponds exactly to the idea in label dependence, where the behaviour of two programs differing only at a single label are compared. In $DTLD$ a statement(expression) is included if replacing it with a different one has an affect on the variables of interest.

There are two major differences between the slices produced by $DTLD$ and those defined by Weiser.

The first is that $DTLD$ does not include statements whose only effect is on the termination conditions of a loop. This means that sometimes the slice produced by $DTLD$ will not terminate in states where the original program terminated. In Figure 6.19(page 198), for example, the $DTLD$ does not include f_5.

The second[6] is that we can have a schema s such that $\neg(xDTVDy)$ and $\neg(xDTVDz)$ but there exist two states differing only on y and x with different non–terminating values for z.

An example is

```
while b1(q)
do while b2(p)
  do begin
        if b3(x)
        then z := f4()
        else p := f5();
         if b6(y)
         then begin
                    x := f7();
                    y := f8()
                 end
          else q := f9()
      end
```

This schema gives rise to the following 'truth table':-

[6] Was only recently noticed. The example is thanks to John Howroyd.

$b1(q)$	$b2(p)$	$b3(x)$	$b6(y)$	Final Value for z
T	T	T	T	$f4()$ or $\perp$
T	T	T	F	$\perp$ (inner loop fails)
T	T	F	T	$\perp$ (outer loop fails)
T	T	F	F	z or $\perp$
T	F			$\perp$ (outer loop fails)
F				z

The values in both column 3 and column 4 must be different to get a different non-$\perp$ value for z.

This means, that surprisingly, the initial values of a set of variables can jointly contribute to the final value of a variable even if they do not contribute individually. The problem is that individually, none of the predicates can affect the final value of z but jointly they can.

In terms of label dependence, '*DTLD* slicing' on z, unlike Weiser slices, would not include statements that jointly affect z in this way.

DTLD of the above schema with respect to z gives the set of labels $\{b1, f4\}$ (see appendix A, page 239, third example).

This cannot happen for *DVD* and *DLD* since $\perp$ in this case is considered a proper value in their case.

Proof: : Suppose $\neg(xDVDy)$ and $\neg(xDVDz)$ but there exist two states, σ_1 and σ_3, say, differing only on y and z with different final values for x. Suppose in σ_1, $(y, z) = (a_1, b_1)$ and in σ_3, $(y, z) = (a_2, b_2)$ where $a_1 \neq a_2$ and $b_1 \neq b_2$ and the values of all other variables agree. Let σ_2 be a state such that $y = a_2$ and every other variable has the same value in σ_1 as in σ_2, then since $\neg(xDVDy)$, σ_1 and σ_2 will result in the same final value of x.

Similarly, since $\neg(xDVDz)$, σ_2 and σ_3 will result in the same final value for x because σ_2 and σ_3 differ only at z. By transitivity, therefore, σ_1 and σ_3 will produce the same final values for x, which is a contradiction.

3.13.2 The Slices produced by *DLD*

We claim that the slice produced by*DLD* is the dataflow minimal version of slicing that preserves the projection, onto the variable of interest, of *standard* semantics, i.e. the slices produced by *DLD* are always executable and always 'behave the same' with respect to the variable of interest: Not only is the value of the variable of interest preserved by the slice, but also the original program and the slice will always 'agree' in terms of termination.

This claim arises from the fact that by Definition 3.9.4(page 103), $l \in$ $DTLD\ x$ if and only if there exist two programs in $[s]$ differing only at l whose standard semantics, $\mathcal{M}$, projected onto x is different.

We further demonstrate the plausibility of our claim with an example.

Consider the schema in Figure 3.23(page 115):-

$$\begin{array}{l} \textit{while}\ f_1(i) \\ \textit{do} \\ \textit{begin} \\ \quad y := f_2(y); \\ \quad i := f_3(i) \\ \textit{end}; \\ z := f_4() \end{array}$$

Figure 3.23: Example

A Weiser slice with respect to z would give $z := f_4()$ as would $DTLD$ whereas a slice that preserves the projection onto variable z of the standard semantics and DLD would give the schema in Figure 3.24(page 115).

$$\begin{array}{l} \textit{while} f_1(i) \\ \textit{do} \\ \textit{begin} \\ \\ \quad i := f_3(i) \\ \textit{end}; \\ z = f_4() \end{array}$$

Figure 3.24: DLD Slice

The reason for this is there are programs in the class of the schema in Figure 3.23(page 115) which do not terminate. These programs terminate if and only if the 'corresponding programs' in the class of the schema in Figure 3.23(page 115) do not terminate. If we remove any more statements from the schema in 3.24 the corresponding programs will behave differently either with respect to the final value of z or to their termination properties.

What we are claiming is now formally stated:-

Claim 3.13.1 *Given any schema s and variable x, the set of labels given by:-*

$$\{l \mid x \ DLD \ l \text{ in } s\}$$

make up a syntactically valid schema s' and what is more, for all $p \in [s]$ the corresponding $p' \in [s']$ is such that for all states σ,

$$\mathcal{M}[\![p]\!]\sigma \ x = \mathcal{M}[\![p']\!]\sigma \ x.$$

It seems intuitively 'correct' that if a label that can have an effect on x on its own, where a label in $DLD \ x$ is left out of a 'potential slice', then there will be a program corresponding to the resulting schema which does not behave the same with respect to the equivalent original program either with respect to x or termination. The proof of this result is left for future work (see Chapter 9).

These slices are larger than those defined by Weiser because Weiser's were not required to behave the same as the original when the original failed to terminate. In general, using this approach, the 'skeletons' of all loops of the program being sliced will be included.

3.14 The Dataflow Minimality of Algorithms for $DTVD$ etc.

For each dataflow dependency introduced in this chapter, if algorithms exist for computing it, it will be dataflow minimal. We prove for example that any algorithm that produces $DTVD$ is dataflow minimal. The proof of each of the others is identical.

3.14.1 Example: DTVD

In an algorithm A for DTVD, the input is a schema s (which is a representation of a control flow graph, so A is a dataflow algorithm) and the output is binary relation d on variables. The ordering is $\subseteq$ and the slice relation R is given by

$R(s, d)$ if and only if

$$d = \{(x, y) | \exists\ p\ \in\ [s] \text{ such that } x\ TVD\ y \text{ in } p\}$$

Proof: Trivial since

$$DTVD(s) = \{(x, y) | \exists\ p\ \in\ [s] \text{ such that } x\ TVD\ y \text{ in } p\}$$

For any schema s, there is trivially, only one set d satisfying the property $R(s, d)$ so, trivially, any algorithm for producing it must be minimal.

3.15 Conclusion

- In this chapter, the dataflow minimality problem has been formally defined.
- We have defined four dataflow dependence relations all acting on a control flow graph or schema. Given a control flow graph g, these dependencies, called *DVD*, DTVD, *DLD* and *DTLD*, have all been defined in terms of the existence of a program p whose control flow graph is g with desired properties. These desired properties have been defined in terms of the standard semantics of p.
- These dependencies, two of which are a form of slicing, are such that if algorithms for computing them exist, then these algorithms must be dataflow minimal.

In the next chapter, we introduce a structure: the Symbolic Execution Tree, which is suitable for expressing these dependencies.

Chapter 4

The Semantics, $\mathcal{S}$, of Loop–free Schemas

4.1 Introduction

The semantics, $\mathcal{S}$, of loop-free schemas is defined as a mapping from loop–free schemas to **symbolic execution trees**.

Symbolic execution trees are finite binary trees whose intermediate nodes are *symbolic predicates* and whose leaf nodes are *symbolic states* which map variable names to symbolic values.

The chapter ends with an implementation of $\mathcal{S}$ in the functional language, Hope [6]. The input to this implementation is a representation of a schema s and the output is a representation of the symbolic execution tree, $\mathcal{S}[\![s]\!]$. This is the first stage in an algorithm for computing the dataflow dependencies introduced in Chapter 3.

4.2 Symbolic Values

Symbolic Values represent the composition of expression functions that would be required to calculate the value of a variable. They are sometimes called *terms* [44].

Definition 4.2.1 (Symbolic Values(Δ))

$$\Delta = V \ \oplus \ (L \times \mathcal{P}\Delta) \ \oplus \ \bot$$

A symbolic value is, thus, one of:

type 1: A variable.

type 2: A label and a finite set of symbolic values.

type 3: Bottom.

4.2.1 Examples of Symbolic Values

- x A single variable. The initial value of variable x.
- $f()$ The label in this case is f, and the set is empty. This is the value of the variable x after the constant assignment $x := f()$.
- $f(x, y, z)$ The label in this case is f, and the set is $\{x, y, z\}$. This is the value of the variable x after the assignment $x := f(x, y, z)$.
- $f(x, g())$ The label in this case is f, and the set is $\{x, g()\}$. This is the value of the variable x after the sequence of assignments $y := g(); x := f(x, y)$.
- $f(x, h(y, z), g(z))$ The label in this case is f, and the set is $\{x, h(y, z), g(z)\}$. This is the value of the variable x after the sequence of assignments

  $$k := g(z); z := h(y, z); x := f(x, z, k).$$

- $\bot$ The bottom symbolic value. As will be seen, the schema *FAIL*, results in a symbolic state where all variables are mapped to this symbolic value.
- $f(\bot, x)$ Symbolic values like this although syntactically valid, never occur as a result of symbolic execution.

Note that a symbolic expression is a special type of symbolic value, where the set component consists just of variables (like the first three examples above).

4.2.2 Symbolic States(Ψ)

As in conventional semantics [88], a state maps variables to values, only in the case of symbolic states, the values are *symbolic values*.

Definition 4.2.2 (Symbolic States)
A symbolic state $\psi \in \Psi$ is a total function from V to Δ.

$$\Psi = [V \rightarrow \Delta]$$

The reason that symbolic states are total is that every variable that is not assigned to in a program is mapped to itself[1] (even variables that are not mentioned at all). This reflects the fact that if a variable is not mentioned in a program fragment, then its final value after executing the fragment will depend on its initial value and nothing else.

4.2.3 Symbolic Execution of a Schema

As an example of how the symbolic state changes as a result of an assignment, consider the simple program, $p_{4.1}$ consisting of a sequence of assignments and its corresponding schema, $s_{4.1}$:-

```
x:=21;
y:=x+5;
z:=x+y
```

$p_{4.1}$

$$x := f_1();$$
$$y := f_2(x);$$
$$z := f_3(x, y)$$

$s_{4.1}$

In Figure 4.1(page 122), we show the steps in the symbolic execution of the sequence of assignments in $s_{4.1}$.

The initial state maps every variable to itself. As in conventional semantics [88], execution of an assignment x:=$f(V)$ (where V is a set of variables) involves updating the current state to reflect the fact that the new value of x is the result of evaluating $f(V)$ in the current state.

[1] This is allowable since variables are symbolic values.

Schema $s_{4.1}$	val of x	val of y	val of z
	x	y	z
$x := f_1();$	$f_1()$	y	z
$y := f_2(x);$	$f_1()$	$f_2(f_1())$	z
$z := f_3(x, y)$	$f_1()$	$f_2(f_1())$	$f_3(f_1(), f_2(f_1()))$

Figure 4.1: The Symbolic execution of schema $s_{4.1}$.

To evaluate $f(V)$ in a symbolic state ψ we simply replace each element of V by its value in ψ (This is '*evaldelta*'– see Definition 4.4.6(page 129)). After the assignment statement $x := f_1()$ the value if x in the state is thus updated to the symbolic value $f_1()$. This reflects the fact that now x depends upon the expression function f_1 which must be a constant function since its set of mentioned variables is empty. After the assignment $y := f_2(x)$, the value of y is updated to the symbolic value $f_2(f_1())$. The variable x in the expression on the right hand side of this assignment to y is replaced by its current value. This shows that y now depends upon the expression functions f_1 and f_2. Notice that z, at this stage, has not been changed, so it still depends only upon its initial value. Finally, after the assignment $z := f3(x, y)$, to find the value for z we replace the variables x and y in the right hand side of the assignment by their current symbolic values to give the symbolic value $f_3(f_1(), f_2(f_1()))$, which shows that z now depends on all three expression functions f_1, f_2 and f_3. In the final symbolic state, all variables in $s_{4.1}$ are mapped to symbolic expressions that mention no variables. This shows that after executing any program in p in $[s_{4.1}]$, all the final values of the variables of p will be independent of the initial values of any variable. For schemas consisting of sequences of assignments, it will be noticed that the variables mentioned in the current symbolic value, δ, of variable v represent the set of variables upon which v is currently dataflow variable dependent. Similarly, the set of labels occurring in δ correspond to the set of labels to which v is currently dataflow label dependent.

4.3 Symbolic Execution Trees

A Symbolic Execution Tree is a binary tree similar to the symbolic execution tree used by Day [30]. The leaves are *symbolic states* and the intermediate nodes are *symbolic values* corresponding to the execution of the predicates of conditionals. The left subtree corresponds to the 'further execution' that occurs if the predicate evaluates to true and the right subtree corresponds to the 'further execution' that occurs if the predicate evaluates to false.

Definition 4.3.1

Symbolic Execution Trees A Symbolic Execution Tree(SET) is an object of type:

$$\mathrm{SET} = \Psi \quad \oplus \quad \mathrm{SET} \times \Delta \times \mathrm{SET}$$

A symbolic execution tree is thus a binary tree whose intermediate nodes are symbolic values and whose leaf nodes are *symbolic states*.

4.3.1 Example of a Symbolic Execution Tree

In Figure 4.2(page 124), an example symbolic execution tree is given.

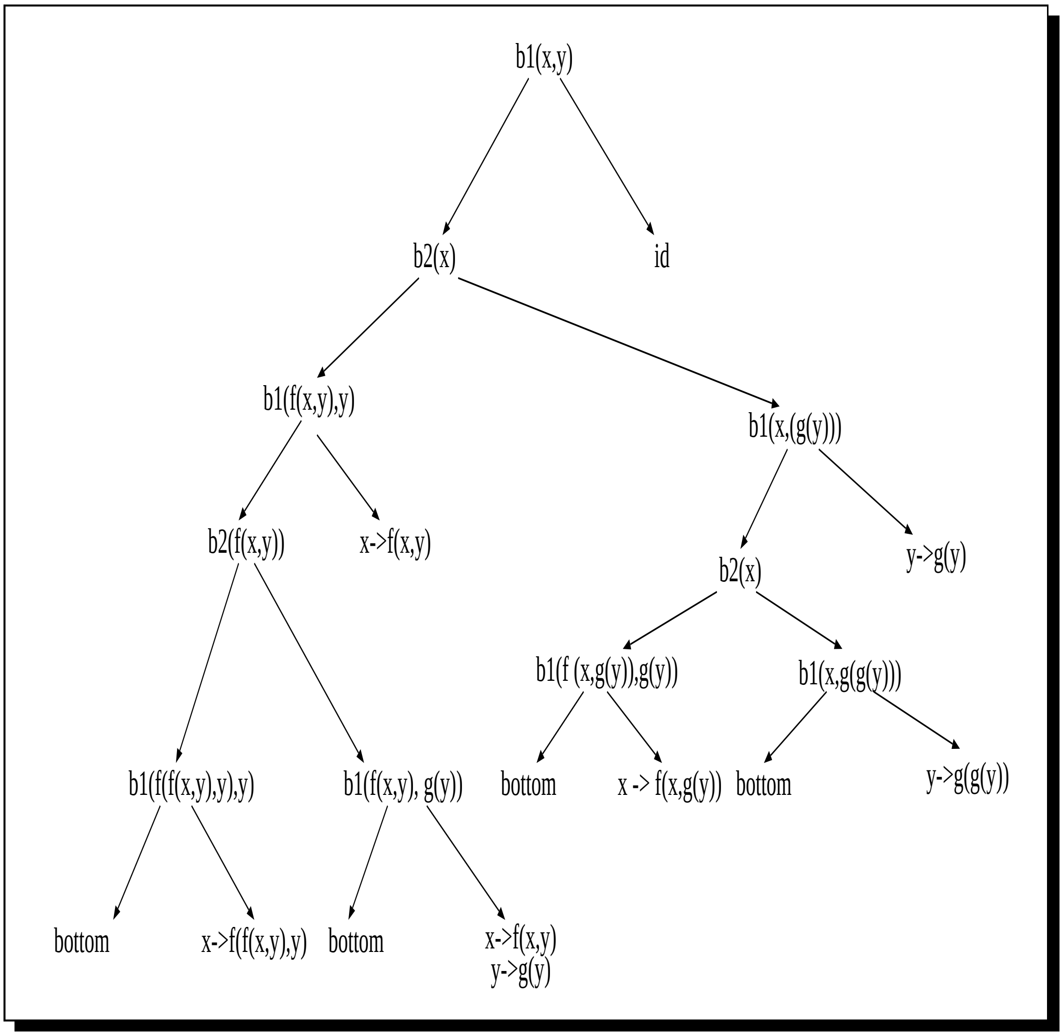


Figure 4.2: Example Symbolic Execution Tree

(It corresponds to schema W_2 defined in Section 7.2.1 page 209.) The intermediate nodes are all symbolic values and the leaf nodes are all final state mapping variables to symbolic values. The value of each variable in these leaf node states is the symbolic value corresponding to the sequence of assignments that would have to be executed to reach that final state. Each intermediate node represents the symbolic execution of a predicate in the schema. We call these nodes *predicate symbolic values*. The *outermost label* of a predicate symbolic value will be the label of the predicate in the schema whose execution this predicate symbolic value represents.

There may be more than one different execution path all leading to the execution of a particular predicate $b(x)$, say. There will be a different occurrence of a predicate symbolic value whose outermost label is b for each execution path that leads to it.

Variables that are not shown in the domain of symbolic states in these diagrams are assumed to be mapped to themselves. There are two 'special' states: The state, id, which maps every variable to itself and the state, $\perp$, where every variable gets mapped to the $\perp$ symbolic value. (Later it will be shown that the $\perp$ symbolic state arises from the schema $FAIL$ that never terminates.)

4.4 Operations on Symbolic Execution Trees

In this section, we define some operations on symbolic execution trees needed for the semantics of loop–free schemas(Section 4.5).

4.4.1 Paths

Definition 4.4.1 (Paths)

Given a symbolic execution tree, t, a *path*, π is a disjoint pair (π_T, π_F) of sets of symbolic values representing a legal path from the root of t to a leaf symbolic state ψ.

π_T and π_F represent the sets of predicate symbolic values that had to be true and false respectively in order to arrive at ψ.

Example

The first two columns in Figure 4.3(page 126) give the values of π_T and π_F respectively for each path of the symbolic execution tree given in Figure 4.4(page 128).

4.4.2 The Path Function, *pfun*, of a symbolic execution tree, t.

The path function, $pfun(t)$, of t is the function whose domain is the set of paths π in t. For each such π, $pfun(t)(\pi)$ is the symbolic state ψ occurring at the leaf of the tree at the end of π. The range of $pfun(t)$ is, thus, the set of all leaves of t. See Figure 4.3(page 126) for an example. It only makes sense to find the path function of *simple* symbolic execution trees(see Section 4.4.3).

True Symbolic Predicates	False Symbolic Predicates	Final State
$\{b1(x,y), b2(x), b1(f(x,y),y), b2(f(x,y)), b1(f(f(x,y),y),y)\}$	$\emptyset$	$\perp$
$\{b1(x,y), b2(x), b1(f(x,y),y), b2(f(x,y))\}$	$\{b1(f(f(x,y),y),y)\}$	$x \mapsto f(f(x,y),y)$
$\{b1(x,y), b2(x), b1(f(x,y),y), b1(f(x,y),g(y))\}$	$\{b2(f(x,y))\}$	$\perp$
$\{b1(x,y), b2(x), b1(f(x,y),y))\}$	$\{b2(f(x,y)), b1(f(x,y),g(y))\}$	$x \mapsto f(x,y)$ $y \mapsto g(y)$
$\{b1(x,y), b2(x)\}$	$\{b1(f(x,y),g(y))\}$	$x \mapsto f(x,y)$
$\{b1(x,y), b1(x,g(y)), b1(x,g(g(y)))\}$	$\{b2(x)\}$	$\perp$
$\{b1(x,y), b1(x,g(y))\}$	$\{b2(x), b1(x,g(g(y))\}$	$y \mapsto g(g(y))$
$\{b1(x,y)\}$	$\{b2(x), b1(x,g(y))\}$	$y \mapsto g(y)$
$\emptyset$	$\{b1(x,y)\}$	id

Figure 4.3: The Path Function of the symbolic execution tree in Figure 4.4(page 128)

Definition 4.4.2 ($pfun : SET \rightarrow (path \nrightarrow \Psi)$)

- If t is a leaf, then

$$pfun\ t = \{(\emptyset, \emptyset) \mapsto t\}$$

- If t is of the form $(t1, r, t2)$, then

$$pfun\ (t1, r, t2) = addleft(r, pfun\ t_1) \cup addright(r, pfun\ t_2)$$

where

$$addleft, addright : (\Delta \times (path \nrightarrow \Psi)) \rightarrow (path \nrightarrow \Psi)$$

$$addleft(r, f) = \bigcup_{(\pi_1,\pi_2) \mapsto \psi \in f} \{(\pi_1 \cup \{r\}, \pi_2) \mapsto \psi\}$$

and

$$addright(r, f) = \bigcup_{(\pi_1,\pi_2) \mapsto \psi \in f} \{(\pi_1, \pi_2 \cup \{r\}) \mapsto \psi\}$$

The path function corresponding to the symbolic execution tree in Figure 4.4(page 128) is given in Figure 4.3(page 126).

4.4.3 Simple Symbolic Execution Trees

A *simple* symbolic execution tree is one where all the paths do not contain repetitions of symbolic values.

Definition 4.4.3 (Simple Symbolic Execution Trees)
Formally, a leaf t is simple,
and the tree $(t1, r, t2)$ is simple if and only if t_1 and t_2 are both simple and r is not a node of t_1 and r is not a node of t_2.

4.4.4 Simplification of a Symbolic Execution Tree

In the semantics of loop free schemas which follows, the meaning of the sequence of two schemas $s_1; s_2$ is defined, unsurprisingly, in terms of the meanings of s_1 and s_2. In the *sequence* function (Definition 4.4.9(page 131), every leaf state ψ in s_1 is replaced by the symbolic execution tree corresponding to the meaning of s_2 evaluated in state ψ using the *treeinstate* function (see Definition 4.4.8(page 130)). The resulting symbolic execution tree constructed in this way may contain 'impossible paths'. An impossible path, is one where π_t and π_f are not disjoint. In order to stop this, we always *simplify* newly created symbolic execution trees. The resulting tree must be simplified to remove these impossible paths.

Example

The symbolic execution tree in Figure 4.2(page 124) is not simple as there is more than one occurrence of the symbolic predicate $b2(x)$. When simplified, the 'lower' occurrence of $b2(x)$ is removed as is everything to the left of this lower occurrence. The simplified symbolic execution tree is shown in Figure 4.4(page 128).

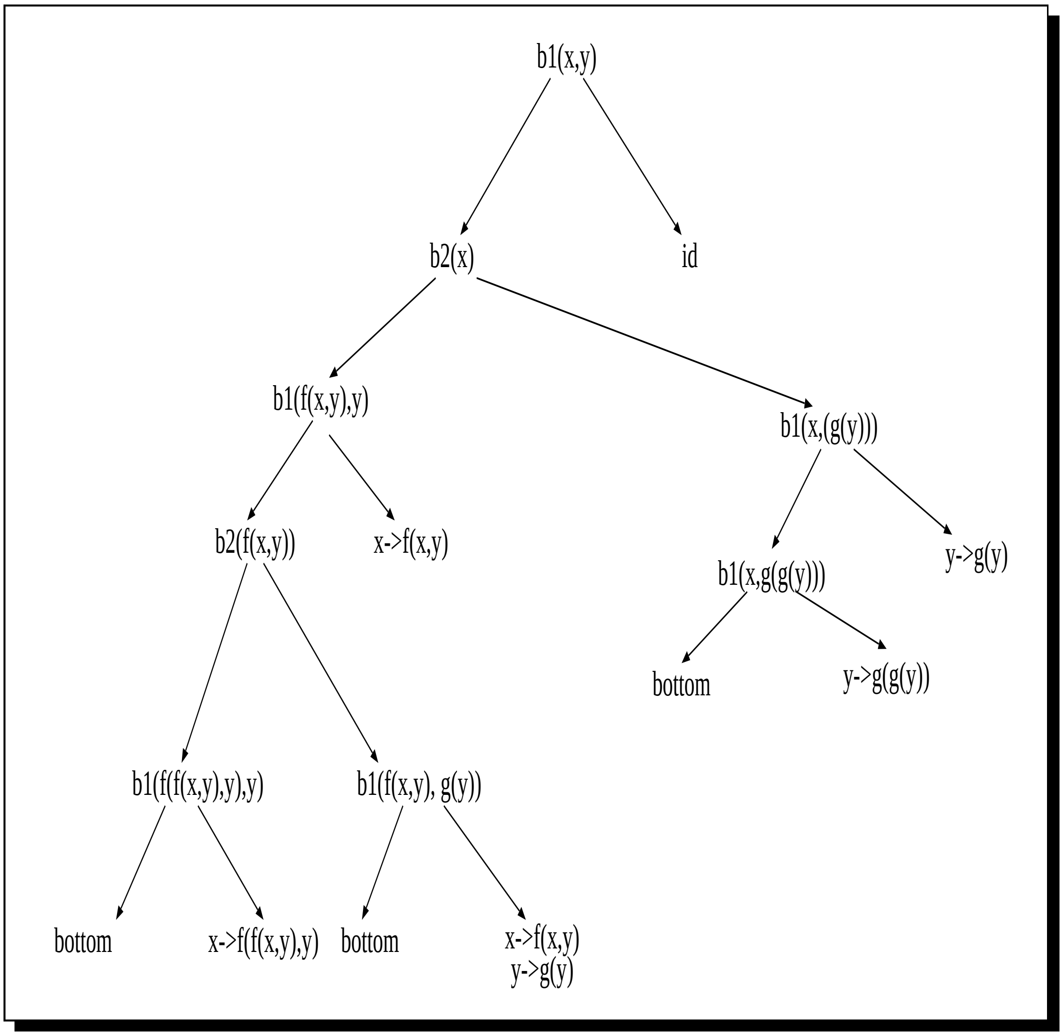


Figure 4.4: Simplified Symbolic Execution Tree

4.4.5 Pruning Symbolic Execution Trees

Simplification is defined in terms of *pruning*. A symbolic execution tree, t, is pruned with respect to a path (π_T, π_F). Pruning a symbolic execution tree with respect to a path (π_T, π_F) both results in a tree which is both simple and contains none of the nodes in $(\pi_T \cup \pi_F)$(Lemma 5.3.1(page 144)). Pruning is defined recursively as follows:-

Definition 4.4.4 ($prune : path \rightarrow SET \rightarrow SET$)
If t is a leaf,

$$prune(\pi_T, \pi_F)(t) = t$$

and if t is of the form (t_1, r, t_2) then,

$$prune(\pi_T, \pi_F)(t) = prune(\pi_T, \pi_F)t1 \quad \text{if } r \in \pi_T$$

$$prune(\pi_T, \pi_F)(t) = prune(\pi_T, \pi_F)t2 \quad \text{if } r \in \pi_F$$

and if $r \notin (\pi_T \cup \pi_F)$ then

$$prune(\pi_T, \pi_F)(t) = (prune(\{r\} \cup \pi_T, \pi_F)t1, r, prune(\pi_T, \{r\} \cup \pi_F)t_2)$$

A symbolic execution tree is, thus, simplified by pruning it with respect to the 'empty' path:

Definition 4.4.5 ($simplify : SET \rightarrow SET \rightarrow SET$)

$$simplify = prune(\emptyset, \emptyset)$$

4.4.6 Evaluating a symbolic value δ in a Symbolic State ψ

The function, *evaldelta*, is equivalent to the $\mathcal{E}$ function in standard semantics. Like, $\mathcal{E}$, which takes expressions and states, *evaldelta* takes the 'symbolic equivalents': symbolic values and symbolic states.

Definition 4.4.6 ($evaldelta : \Psi \rightarrow \Delta \rightarrow \Delta$)
If δ is a variable x, then

$$evaldelta\ \psi\ x = \psi\ x$$

If δ is of the form $f(S)$, then

$$evaldelta\ \psi\ \delta = f(\bigcup_{s \in S} evaldelta\ \psi\ s)$$

evaldelta is strict in both its arguments:-

$$evaldelta\ \psi\ \bot = \bot = evaldelta\ \bot\ \psi$$

Example

See Section 4.2.3 which shows *evaldelta* being applied.

4.4.7 Updating a Symbolic State in a Symbolic State

Given two symbolic states ψ_1 and ψ_2, updating ψ_2 in state ψ_1 means producing the state which maps each variable, v, to the result of evaluating the symbolic value $\psi_2\ x$ in ψ_1.

Definition 4.4.7 ($updatestateinstate : \Psi \to \Psi \to \Psi$)

$$updatestateinstate\ \psi_1\ \psi_2\ x = evaldelta\ \psi_1\ (\psi_2\ x)$$

updatestateinstate is strict in both its arguments:-

$$updatestateinstate\ \psi\ \bot = \bot = updatestateinstate\ \bot\ \psi$$

4.4.8 Evaluating a Symbolic Execution Tree in a Symbolic State

To evaluate *treeinstate* $t\ \psi$, we symbolically evaluate each node of t in state ψ. The intermediate nodes are symbolic values so we use *evaldelta* for these and the leaf nodes are states so we use *updatestateinstate* for these.

Definition 4.4.8 ($treeinstate : \Psi \to SET \to SET$)
If t is a leaf, then

$$treeinstate\ \psi\ t = updatestateinstate\ \psi\ t$$

and if t of the form $(t1, r, t2)$, then

$$treeinstate\ \psi\ t$$
$$=$$
$$(treeinstate\ \psi\ t_1, evaldelta\ \psi\ r, treeinstate\ \psi\ t_2)$$

4.4.9 The Sequence of two Symbolic Execution Trees

Given two symbolic execution trees, t and t' we now define the meaning of $t; t'$ i.e. t followed by t'. To compute $t; t'$ we replace each leaf node ψ of t by treeinstate t' ψ.

Definition 4.4.9 ($sequence : SET \to SET \to SET$)
If t is a leaf,

$$sequence\ t\ t' = treeinstate\ t\ t'$$

and if t is of the form (t_1, r, t_2) then,

$$sequence\ t\ t' = (sequence\ t_1\ t', r, sequence\ t_2\ t').$$

4.5 The Semantics of Loop Free Schemas

We are now in a position to define the semantic function $\mathcal{S}$ which maps schemas which do not contain while loops to symbolic execution trees. In effect, $\mathcal{S}$ is an algorithm which translates schemas into symbolic execution trees. Later, the resulting symbolic execution tree will be further analysed to produce the various dataflow dependencies introduced in Chapter 3.

In the following chapter(Chapter 6), the algorithm will be proved correct for loop free schemas.

4.5.1 Assignments

Definition 4.5.1 (Assignment)
$\mathcal{S}[\![x{:=}f(V)]\!]$ is the symbolic execution tree consisting of the single leaf state:-

$$\mathcal{S}[\![x{:=}f(V)]\!] = \lambda z. \begin{cases} z & \text{if } z \neq x \\ f(V) & \text{if } z = x \end{cases}$$

4.5.2 Fail

Definition 4.5.2 (*FAIL*)

FAIL is a schema that represents programs which never terminate in any state. $\mathcal{S}[\![FAIL]\!]$ is thus the symbolic execution tree consisting of the single leaf $\perp$.

$$\mathcal{S}[\![FAIL]\!] = \perp$$

4.5.3 Skip

Definition 4.5.3 (*skip*)

skip is a schema that represents a program which does nothing.

$$\mathcal{S}[\![skip]\!] = \lambda v.v$$

4.5.4 Conditionals

Definition 4.5.4 (Conditionals)

$$\mathcal{S}[\![\mathit{if}\ f(V)\ \mathit{then}\ s_1\ \mathit{else}\ s_2]\!] = \mathit{simplify}\ (\mathcal{S}[\![s_1]\!], f(V), \mathcal{S}[\![s_2]\!])$$

$\mathcal{S}[\![\mathit{if}\ f(V)\ \mathit{then}\ s_1\ \mathit{else}\ s_2]\!]$ is thus the symbolic execution tree whose root is the symbolic value, $f(V)$, and whose left and right subtrees are the meanings of the *then* and the *else* branches respectively.

4.5.5 Statement Sequences

Definition 4.5.5 (Statement Sequences)

$$\mathcal{S}[\![s_1; s_2]\!] = \mathit{simplify}\ (\mathit{sequence}\ \mathcal{S}[\![s_1]\!]\ \mathcal{S}[\![s_2]\!])$$

4.5.6 Example

(The sequence of assignments in Figure 4.1(page 122) were really translated to a symbolic execution tree consisting of a single leaf node containing the final state shown in Figure 4.1(page 122).)

Consider the program $p_{4.5}$, and its corresponding schema $s_{4.5}$, in Figure 4.5(page 133). The symbolic execution tree corresponding to $s_{4.5}$ has structure as shown in Figure 4.6(page 134).

```
if i<3
then
  begin
   c:=k;
   if c=2
   then c:=y
   else c:=c+1;
   i:=i+1
  end
else
 skip
```

```
if f1(i)
then
  begin
   c := f2(k);
   if f3(c)
   then c := f4(y)
   else c := f5(c);
   i := f6(i)
  end
else
 skip
```

Figure 4.5: $p_{4.5}$ and $s_{4.5}$

Nodes A and B are symbolic values corresponding to the evaluation of the symbolic predicates $f1(i)$ and $f3(c)$ respectively. The leaves C, D and E of the tree correspond to symbolic states produced by symbolically executing the sequences of assignments along that path as described in the previous section. These final states are dependent on the values of the two symbolic predicates(Left branch=*true*, right branch=*false*).

4.6 Implementation of the Semantics of Loop Free Schemas

In this section, the semantics described in this chapter is translated into an algorithm in the functional programming language Hope [6]. The input to the algorithm (given by the function `meaningl` below) is a representation of a schema (in abstract syntax[2]) and the output is a

[2] We have not included the parser as it does not contribute to the understanding of the algorithm.

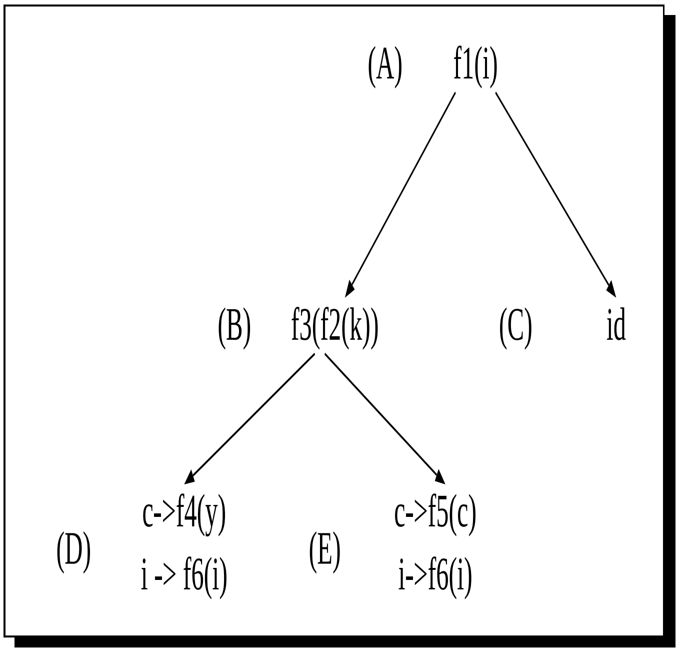


Figure 4.6: The symbolic execution tree of $s_{4.5}$

representation of the corresponding symbolic execution tree. -verb-delta- is the Hope data type representing symbolic values.

Since Hope is a functional language, it is noticed that the program definitions are only superficially different from the mathematical ones introduced in this chapter. This means that the inductive proofs in terms of the mathematical definitions, occurring in later chapters, will also serve as proofs of the program itself.

4.6.1 The Abstract Syntax for Symbolic Values (Definition 4.2.1(page 119))

```
type name == list(char);
data  delta ==  va name ++ complex (name # (set delta)) ++ botdelta;
```

4.6.2 The 'Standard' Update Function [88]

```
update: (alpha -> beta) -> alpha -> beta -> (alpha -> beta);
update f x y z <= if z=x
                  then y
                  else f z;
```

4.6.3 The Abstract Syntax for Schemas (Section 3.11).

```
data statement ==
 FAIL ++
 ass(name X delta) ++
 ife(delta X (list statement) X (list statement)) ++
 while(delta X (list statement));
```

4.6.4 Symbolic States (Section 4.2.2)

```
data state== ok(name -> delta) ++ botstate;
```

4.6.5 The Abstract Syntax for Symbolic Execution Trees. (Section 4.3)

```
data SET == leaf state   ++  node(SET X delta X SET);
```

4.6.6 Representation of Paths (Definition 4.4.1(page 125))

```
type path == set delta X set delta;
```

4.6.7 *evaldelta* (Definition 4.4.6(page 129))

```
evaldelta: state -> delta -> delta;
evaldelta botstate x <= botdelta;
evaldelta (ok sigma) botdelta <= botdelta;
evaldelta (ok sigma) (va x) <=   sigma x;
evaldelta (ok sigma) (complex (f,S)) <=
complex(f,mapset1(evaldelta (ok sigma) ,S));
```

4.6.8 *updatestateinstate* (Definition 4.4.7(page 130))

```
updatestateinstate:state -> state -> state;
updatestateinstate (ok st1)  (ok st2)  <= ok((evaldelta (ok st1) o st2));
updatestateinstate x y <= botstate;
```

4.6.9 *treeinstate* (Definition 4.4.8(page 130))

```
treeinstate:  SET ->state -> SET;

treeinstate (leaf sigma') sigma <= leaf (updatestateinstate sigma sigma');

treeinstate (node(t1,r,t2)) sigma  <=

node (treeinstate t1 sigma,evaldelta sigma r, treeinstate t2 sigma);
```

4.6.10 *sequence* (Definition 4.4.9(page 131))

```
sequence:SET -> SET -> SET;

sequence (leaf sigma) t'  <=  treeinstate t' sigma;

sequence(node(t1,r,t2)) t' <=  node(sequence t1 t',r,sequence  t2 t');
```

4.6.11 *prune* (Definition 4.4.4(page 128))

```
prune: path -> SET -> SET;

prune (l,m) (leaf x) <= leaf x;

prune (l,m) (node(b1,r,b2)) <=

if (r isin l)

then prune (l,m) b1

else if (r isin m)

     then prune (l,m) b2

     else node(prune (r & l,m) b1, r, prune (l,r & m) b2);
```

`x & y` is Hope notation for the set $\{x\} \cup y$

4.6.12 *simplify* (Definition 4.4.5(page 129))

```
simplify: SET -> SET;

simplify <= prune(empty,empty);
```

4.6.13 The Semantic Function $\mathcal{S}$ (Section 4.5)

```
meaning:statement -> SET;

meaningl:list(statement) -> SET;
```

'`meaning`' is the semantic function $\mathcal{S}$ (Section 4.5)

4.6.14 The *skip* Rule (Definition 4.5.3(page 132))

```
meaningl nil  <= leaf (ok va);
```

4.6.15 The Sequence Rule (Definition 4.5.5(page 132))

```
meaningl (x::l)  <= simplify (sequence (meaning x) (meaningl l));
```

4.6.16 The *FAIL* Rule (Definition 4.5.2(page 132))

```
meaning FAIL <= leaf botstate;
```

4.6.17 The Assignment Rule (Definition 4.5.1(page 131))

```
meaning (ass(x,e))   <=
leaf(ok (update va x (evaldelta (ok va) e)));
```

4.6.18 The Conditional Rule (Definition 4.5.4(page 132))

```
meaning (ife(e,l1,l2))  <=
         simplify (node(meaningl l1, evaldelta  (ok va) e,meaningl l2));
```

4.6.19 The Path Function (Definition 4.4.2(page 125))

```
singleton: alpha -> set alpha;
singleton x <= x & empty;


dec addleft,addright: delta X (pfun path delta) -> (pfun path delta);
addleft(d,f) <= mapset(lambda ((a,b),c) => singleton((d & a,b),c),f);


addright(d,f) <= mapset(lambda ((a,b),c) => singleton((a,d & b),c),f);


applystate :name -> state -> delta;
applystate v (ok sigma) <= sigma v;
applystate v botstate <= botdelta;
```

```
pathfun: SET -> name  -> (pfun path delta);
pathfun (leaf  sigma) v <=  singleton((empty,empty),applystate v sigma);
pathfun (node (b1,r,b2)) v <=  addleft (r,pathfun b1 v) U addright (r,pathfun b2 v);
```

4.7 Conclusion

The semantics, $\mathcal{S}$, of loop-free schemas has been defined as a mapping from loop–free schemas to symbolic execution trees. An implementation of $\mathcal{S}$ has been given. This is the first stage in an algorithm for computing the dataflow dependencies introduced in Chapter 3.

In order to be able to prove these algorithms correct, it will first be necessary to prove that $\mathcal{S}$ is both sound and complete. This is done in Chapter 5.

Chapter 5

The Soundness and Completeness of $\mathcal{S}$

5.1 Introduction

In this chapter, it is shown how for each loop free schema, s, the symbolic execution tree, $\mathcal{S}[\![s]\!]$, characterises the set of all possible behaviours of all programs in $[s]$.

The reason that this is critically important, is that without it, there is no justification that it is valid to infer properties of a schema, and hence of the set of programs that the schema represents, by analysis of its symbolic execution tree.

The theory in this chapter leads to a proof that the characterisation provided by $\mathcal{S}$ is both sound and complete.

- $\mathcal{S}$ is complete in the following sense:
  Given a loop–free schema s, and a program $p \in [s]$, and a state, σ, there exists exactly one path π, of the symbolic execution tree, $\mathcal{S}[\![s]\!]$, that corresponds to the execution of p in state σ.

- $\mathcal{S}$ is sound in the following sense:
  For all paths π, of the symbolic execution tree, $\mathcal{S}[\![s]\!]$, there exists a program, $p \in [s]$, and a state, σ, such that π corresponds to the execution of p in state σ.

It is this theorem that shows that the symbolic execution tree captures exactly the right semantic information about the set of programs represented by the schema; no more and no less. It is because of this, that program analysis using symbolic execution trees may have other applications, not just the ones introduced in this thesis. It is this theorem which provides the semantic link that enables the algorithms given in Chapter 6, for computing dataflow dependencies of loop–free schemas (defined in terms of their symbolic execution trees) to be proved correct.

5.2 The Correspondence between Symbolic Execution Trees and Programs

The path π, of a symbolic execution tree *corresponds* to the execution of program, p, in state, σ, if and only if:-

- π is *satisfied* by p and σ (Definition 5.2.2(page 143)),

  and

- the state *derived* (see Section 5.2.2) from the symbolic state at 'the end of' the path π with respect to s, p and σ is $\mathcal{M}[\![p]\!]\sigma$.

5.2.1 The Function *evalsym*

Given a program, p in the class $[s]$, represented by schema, s, and a starting state, σ, every symbolic value, δ, that can arise in the symbolic execution of s corresponds to a 'real' value,

$$evalsym\ s\ p\ \sigma\ \delta$$

that arises in the execution of p, starting in state sigma. The function, *evalsym*, is the one that computes a value (not symbolic, but real) from a symbolic value. In order to do this we need two things:

1. A program, p, which tells us which 'real expressions' to use in place of the terms of the symbol value that we are evaluating

2. and a state, σ, in which to evaluate the resulting 'real expression'.

Including the schema, s, as a parameter to *evalsym* is not strictly necessary.

For each symbolic expression, $f_i(V_i)$ in s there is a corresponding expression, e_i in p (see Section 3.11.3). Since $p \in [s]$, e_i is not affected by any variable outside V_i. So $\mathcal{E}[\![e_i]\!]\sigma$ can be computed if we know the values in σ of all the variables in V_i.

Non–trivial symbolic values consist of a label and a *set* of symbolic values. As described in Section 3.11.3, the label refers to the expression function in a program in $[s]$. The set of symbolic values inside a non–trivial symbolic value correspond to the values that are passed to the expression function. How do we know which symbolic value in this set is associated with which variable? This is not a problem, since labels are unique. If the outermost label of symbolic value, f_i, occurs as the label of an expression in an assignment to x, say, then, by uniqueness (Section 3.11.2), it cannot occur as the outermost label of an assignment to any other variable[1]. The value of all symbolic values whose outermost label is f_i must be therefore associated with the variable, x, and only the variable x. Given a schema, s, and symbolic value $f_i(S)$ we define $varof(s, f_i(S))$ to be the variable associated with f_i as just defined. For example, in the schema in Figure 3.22(page 109), $varof(f_3(y)) = c$.

For a trivial symbolic value, i.e. a variable v, $varof(s, v) = v$.

To evaluate a symbolic value in terms of a schema s and an initial state σ can thus be defined as follows:

Definition 5.2.1 (evalsym)
The function, *evalsym*, has the type given by:-

$$evalsym : \text{Schemas} \nrightarrow \text{Programs} \nrightarrow \text{States} \nrightarrow \text{Symbolic Values} \nrightarrow \text{Values}$$

- If x is a variable,

$$evalsym\ s\ p\ \sigma\ x = \sigma x,$$

[1] It is for this reason that we can define symbolic value in terms of sets rather than lists of symbolic values.

- for compound symbolic values, $f_i(S)$,

$$evalsym\ s\ p\ \sigma\ f_i(S) = \mathcal{E}[\![p\ f_i]\!] \bigcup_{\delta \in S} varof(s, \delta) \mapsto (evalsym\ s\ p\ \sigma\ \delta),$$

  where the $p\ f_i$ are the expressions in program p corresponding to the f_i in s

- and for the $\perp$ symbolic value,

$$evalsym\ s\ p\ \sigma\ \perp = \perp.$$

$\bigcup_{\delta \in S} \{varof(s, \delta) \mapsto (evalsym\ s\ p\ \sigma\ \delta)\}$ defines the state function[2] in which to evaluate the expression $p\ f_i$.

It should be noted that *varof* is a well defined function, since for any symbolic value $f_i(S)$that can arise,

$$\delta_1 \in S \text{ and } \delta_2 \in S \text{ and } \delta_1 \neq \delta_2 \implies varof(s, \delta_1) \neq varof(s, \delta_2).$$

We call $evalsym\ s\ p\ \sigma\ \delta$ the *derived value of the symbolic value δ with respect to (s, p, σ)*.

5.2.2 The Derived State

Given a program, p in the class $[s]$, represented by schema, s and a starting state σ, every symbolic state, ψ that can arise in the symbolic execution of s thus corresponds to a 'real' state,

$$\lambda v.evalsym\ s\ p\ \sigma\ (\psi\ v).$$

We call this the *derived state of the symbolic state ψ with respect to* (s, p, σ).

5.2.3 The function *satisfy*

A program p satisfies a path, π in a state σ if and only if executing p in state σ 'gives rise' to the path π. Formally:

[2]Here we are representing a state function in the standard set theoretic manner, as a set of variable $\mapsto$ value pairs.

Definition 5.2.2 (*satisfy*)

Given a schema s and a path $\pi = (\pi_t, \pi_f)$ a program $p \in [s]$ and a state σ

$$\begin{gathered} \mathit{satisfy}\ s\ p\ \sigma\ \pi \\ \Longleftrightarrow \\ \text{for all } \delta \text{ in } \pi_t,\ \mathit{evalsym}\ s\ p\ \sigma\ \delta = \mathit{true} \\ \text{and} \\ \text{for all } \delta \text{ in } \pi_f,\ \mathit{evalsym}\ s\ p\ \sigma\ \delta = \mathit{false}. \end{gathered}$$

The definition of 'satisfy' is extended to symbolic execution trees as follow:

Definition 5.2.3 (Satisfying a Symbolic Execution Tree)

Given a schema s and a path $\pi = (\pi_t, \pi_f)$ a program $p \in [s]$, a symbolic execution tree, t, and a state σ

$$\begin{gathered} \mathit{satisfy}\ t\ p\ \sigma\ \pi \\ \Longleftrightarrow \\ \pi \in \mathit{dom}(\mathit{pfun}\ t) \text{ and } \mathit{satisfy}\ s\ p\ \sigma\ \pi. \end{gathered}$$

5.2.4 Differences

Given two paths, the *differences* between them are simply the set of symbolic values that are true in one path and false in the other.

Definition 5.2.4 (*diffs*)

Let (π_1, π_2) and (π'_1, π'_2) be paths.
$\mathit{diffs}((\pi_1, \pi_2), (\pi'_1, \pi'_2)) = (\pi_1 \cap \pi'_2) \cup (\pi_2 \cap \pi'_1)$.

5.3 Further Results

The results of this section are mainly technical confirmations that the definitions in Chapter 4 are correct. They are all needed in the proof of the main theorem of this chapter, Theorem 5.4.1(page 160).

5.3.1 The Result of Pruning a Simple Symbolic Execution Tree is Simple

It is now proved that pruning a simple tree with respect to a path π results in a tree that is still simple and that none of the elements of π are in the pruned tree.

Lemma 5.3.1 *If t is simple then*

1. *$prune(\pi_t, \pi_f)\ t$ is simple and*

2. *No element of $\pi_t \cup \pi_f$ is a node of $prune(\pi_t, \pi_f)\ t$.*

Proof: By induction on the depth of t.

Base Case If t is a leaf, then $prune(\pi_t, \pi_f)\ t = t$ and t is simple.

Let $t = (t_1, r, t_2)$ be simple.

Consider $prune(\pi_t, \pi_f)\ (t_1, r, t_2)$.

If $r \in \pi_t$ then, by Definition 4.4.4(page 128), $prune(\pi_t, \pi_f)\ (t_1, r, t_2) = prune(\pi_t, \pi_f)\ (t_1)$ which is simple by induction hypothesis.

Similarly, if $r \in \pi_f$ then, by Definition 4.4.4(page 128), $prune(\pi_t, \pi_f)\ (t_1, r, t_2) = prune(\pi_t, \pi_f)\ (t_2)$ which is simple by induction hypothesis.

If $r \notin \pi_t \cup \pi_f$, then, by Definition 4.4.4(page 128),

$prune(\pi_t, \pi_f)\ (t_1, r, t_2) = (prune(\{r\} \cup \pi_t, \pi_f)t_1, r, prune(\pi_t, \{r\} \cup \pi_f)\ t_2)$.

But, by induction hypothesis, $prune(\{r\} \cup \pi_t, \pi_f)t1$ and $prune(\pi_t, \{r\} \cup \pi_f)\ t_2$ are simple and do not contain $\pi_t \cup \pi_f \cup \{r\}$, so by Definition 4.4.3(page 127),

$(prune(\{r\} \cup \pi_t, \pi_f)t_1, r, prune(\pi_t, \{r\} \cup \pi_f)t_2)$ is simple and does not contain $\pi_t \cup \pi_f$.

5.3.2 The Result of Simplifying a Symbolic Execution Tree is Simple

We now prove that simplifying a symbolic execution tree results in a simple symbolic execution tree.

Theorem 5.3.1 *For all symbolic execution trees, t, $(simplify\ t)$ is simple.*

Proof: By induction on the depth of t.

Base Case If t is a leaf, $(simplify\ t) = t$ which is simple.

Let $t = (t_1, r, t_2)$.

$simplify(t_1, r, t_2) = prune(\emptyset, \emptyset)(t_1, r, t_2) = (prune(\{r\}, \emptyset)t_1, r, prune(\emptyset, \{r\})t_2)$.

By the Lemma 5.3.1(page 144) and the induction hypothesis, $prune(\{r\}, \emptyset)t_1$ and $prune(\emptyset, \{r\})t_2$

are simple and do not contain r. So $(prune(\{r\},\emptyset)t_1, r, prune(\emptyset,\{r\})t_2)$ is simple as required, by induction.

5.3.3 A Partial Order on Paths

We now define what it means for one path to be 'less' than another.

Definition 5.3.1 ($\pi \sqsubseteq \pi'$)
$\pi \sqsubseteq \pi' \iff \pi_t \subseteq \pi'_t$ and $\pi_f \subseteq \pi'_f$.

Informally, $\pi \sqsubseteq \pi'$ means that

- every symbolic predicate that is true in π is also true in π'
  and
- every symbolic predicate that is false in π is also false in π'.

In other words, π can be obtained from π' by *deleting* symbolic predicates. Clearly, given a symbolic execution tree, t, $\sqsubseteq$ defines a partial order on the set of all paths of t.

5.3.4 'Smaller Path' Lemma

The next lemma shows that every path of a pruned symbolic execution tree is less than some path of the unpruned tree, i.e. each path of *prune* π t can be obtained by deleting elements from some path of t.

Lemma 5.3.2 *Let t be a simple Symbolic Execution Tree.*
For all paths, π,

$$\pi' \in dom(pfun(prune\ \pi\ t)) \implies \exists \pi'' \in dom(pfun\ t) \text{ such that } \pi' \sqsubseteq \pi'' \text{ and}$$
$$pfun(prune\ \pi\ t)\pi' = pfun\ t\ \pi''.$$

Proof: Induction on the depth of t.

<u>Base Case</u>

If t is a leaf ψ then result follows trivially.

Now let $t = (t_1, r, t_2)$ and $\pi' \in dom(pfun(prune\ \pi\ (t_1, r, t_2)))$.

<u>Case</u>1 If $r \in \pi_t$

then, by Definition 4.4.4(page 128),

$$prune\ \pi\ (t_1, r, t_2) = prune\ \pi\ t_1.$$

Therefore

$$\pi' \in dom(pfun(prune\ \pi\ t_1)).$$

By induction hypothesis,

$$\exists \pi'' \in dom(pfun\ t_1) \text{ such that } \pi' \sqsubseteq \pi'' \text{ and } pfun(prune\ \pi\ t_1)\pi' = pfun\ t_1\ \pi''.$$

Put

$$\pi''' = (\pi''_t \cup \{r\}, \pi''_f).$$

Then

$$\pi''' \in dom(pfun\ (t_1, r, t_2)).$$

Also

$$\pi' \sqsubseteq \pi'''$$

and, since $r \in \pi_t$,

$$pfun(prune\ \pi\ (t_1, r, t_2))\pi' = pfun(prune\ \pi\ t_1)\pi' = pfun\ t_1\ \pi'' = pfun(t_1, r, t_2)\pi'''$$

as required.

Case2 If $r \in \pi_f$

Symmetrical Proof.

Case3 If $r \notin \pi$ then, by Definition 4.4.4(page 128),

$$prune\ \pi\ (t_1, r, t_2) = (prune\ (\pi_t \cup \{r\}, \pi_f)t_1, r, prune\ (\pi_t, \pi_f \cup \{r\})t_2)$$

Therefore by Definition 4.4.2(page 125),

$$pfun(prune\ \pi\ (t_1, r, t_2)) =$$
$$addleft(r, pfun(prune\ (\pi_t \cup \{r\}, \pi_f)t_1) \cup addright(r, pfun(prune\ (\pi_t, \pi_f \cup \{r\})t_2)).$$

Suppose, first, that

$$\underline{\pi' \in dom(addleft(r, pfun(prune\ (\pi_t \cup \{r\}, \pi_f)t_1))).}$$

$$\text{Then } \pi' = (\pi''_t \cup \{r\}, \pi''_f) \text{ for some } \pi'' \in dom(pfun(prune\ (\pi_t \cup \{r\}, \pi_f)t_1)).$$

By induction hypothesis,

$$\exists \pi''' \in dom(pfun\ t_1) \text{ such that } \pi'' \sqsubseteq \pi''' \text{ and}$$
$$pfun(prune\ \ (\pi_t \cup \{r\}, \pi_f)\ t_1)\pi'' = pfun\ t_1\ \pi'''.$$

But

$$\pi' \sqsubseteq (\pi'''_t \cup \{r\}, \pi'''_f).$$

By Definition 4.4.2(page 125),

$$(\pi'''_t \cup \{r\}, \pi'''_f) \in dom(pfun\ (t_1, r, t_2))$$

$$\text{Therefore } pfun(prune\ \pi\ (t_1, r, t_2))\pi' = pfun(prune\ (\pi_t \cup \{r\}, \pi_f)t_1)\pi'' = pfun\ t_1\ \pi'''$$
$$=$$
$$pfun(t_1, r, t_2)(\pi'''_t \cup \{r\}, \pi'''_f) \text{ as required.}$$

Again, by symmetry, we omit the proof of the case

$$\pi' \in dom(addright(r, pfun(prune\ (\pi_t, \pi_f \cup \{r\})t_2))).$$

This completes the proof of Lemma 5.3.2(page 145).

Lemma 5.3.3 $\pi' \sqsubseteq \pi \implies diffs(\pi, \pi') = \emptyset$

Proof: trivial.

5.3.5 'Pruning' Lemma

The next lemma states that given two paths π and π', with no differences and a simple tree t, with π a path of t, if t is pruned with respect to π', then the path $(\pi_t - \pi'_t, \pi_f - \pi'_f)$ will be a path of the resulting tree and what is more, the symbolic state at the end of this path in the pruned tree will be the same symbolic state that occurs at the end of path π in the original tree t.

Lemma 5.3.4 *Let t be a simple symbolic execution tree.*
For all paths π in $dom(pfun\ t)$, for all paths π' with

$$diffs(\pi, \pi') = \emptyset,$$

$$(\pi_t - \pi'_t, \pi_f - \pi'_f) \in dom(pfun\ (prune\ \pi'\ t))$$
$$\textit{and}$$
$$pfun\ t\ \pi = pfun\ (prune\ \pi'\ t)\ (\pi_t - \pi'_t, \pi_f - \pi'_f).$$

Proof: Induction on the depth of t.

<u>Base Case</u> trivial

<u>Induction Hypothesis</u>

Let $t = (t_1, r, t_2)$ be simple.

Assume that for all paths π in $dom(pfun\ t)$, for all paths π' with

$$diffs(\pi, \pi') = \emptyset,$$

$$(\pi_t - \pi'_t, \pi_f - \pi'_f) \in dom(pfun\ (prune\ \pi'\ t_1))$$
$$\text{and}$$
$$pfun\ t_1\ \pi = pfun\ (prune\ \pi'\ t_1)(\pi_t - \pi'_t, \pi_f - \pi'_f)$$

and for all paths π in $dom(pfun\ t)$, for all paths π' with

$$diffs(\pi, \pi') = \emptyset,$$

$$(\pi_t - \pi'_t, \pi_f - \pi'_f) \in dom(pfun\ (prune\ \pi'\ t_2))$$
$$\text{and}$$
$$pfun\ t_2\ \pi = pfun\ (prune\ \pi'\ t_2)(\pi_t - \pi'_t, \pi_f - \pi'_f)$$

Let $\pi \in dom(pfun\ (t_1, r, t_2))$ and π' be a path such that $diffs(\pi, \pi') = \emptyset$.
We must show

$$(\pi_t - \pi'_t, \pi_f - \pi'_f) \in dom(pfun\ (prune\ \pi'\ (t_1, r, t_2)))$$
and
$$pfun\ (t_1, r, t_2)\pi = pfun\ (prune\ \pi'\ \ (t_1, r, t_2))(\pi_t - \pi'_t, \pi_f - \pi'_f),$$

By Definition 4.4.2(page 125),

$$\pi \in dom(addleft(r, pfun\ t_1)) \cup dom(addright(r, pfun\ t_2)).$$

Suppose first, that $\underline{\pi \in dom(addleft(r, pfun\ t_1))}$.
So $\pi = (\pi''_t \cup \{r\}, \pi''_f)$ for some π'' in $dom(pfun\ t_1))$ with $r \notin \pi''$, since t is simple.
There are two possibilities for π':

1. Either $r \in \pi'_t$ or
2. $r \notin \pi'$ (r cannot be in π'_f since $diffs(\pi, \pi') = \emptyset$).

First, suppose $r \in \pi'_t$.
Then by Definition 4.4.4(page 128),

$$dom(pfun\ (prune\ \pi'\ (t_1, r, t_2))) = dom(pfun\ (prune\ \pi'\ t_1)).$$

By induction hypothesis, since $\pi'' \in dom(pfun\ t_1))$ and $diffs(\pi'', \pi') = \emptyset$,

$$(\pi''_t - \pi'_t, \pi''_f - \pi'_f) \in dom(pfun\ (prune\ \pi'\ t_1)) = dom(pfun\ (prune\ \pi'\ (t_1, r, t_2)))$$
and
$$(pfun\ t_1)\pi'' = (pfun\ (prune\ \pi'\ t_2))(\pi''_t - \pi'_t, \pi''_f - \pi'_f).$$

But since $r \in \pi'_t$,

$$(\pi''_t - \pi'_t, \pi''_f - \pi'_f) = (\pi_t - \pi'_t, \pi_f - \pi'_f).$$

Therefore

$$(\pi_t - \pi'_t, \pi_f - \pi'_f) \in dom(pfun\ (prune\ \pi'\ (t_1, r, t_2)))\ \text{as required}$$

and, by Definition 4.4.2(page 125), since $r \in \pi_t$,

$$pfun\ (t_1, r, t_2)\ \pi = pfun\ t_1\ \pi''$$
$$= pfun\ (prune\ \pi'\ t_1)(\pi''_t - \pi'_t, \pi''_f - \pi'_f)\ \text{(by induction hypothesis)}$$
$$=$$
$$pfun\ (prune\ \pi'\ (t_1, r, t_2))(\pi''_t - \pi'_t, \pi''_f - \pi'_f)\ (\text{since } r \in \pi'_t)$$
$$=$$
$$pfun\ (prune\ \pi'\ (t_1, r, t_2))(\pi_t - \pi'_t, \pi_f - \pi'_f)$$
$$(\text{since } (\pi''_t - \pi'_t, \pi''_f - \pi'_f) = (\pi_t - \pi'_t, \pi_f - \pi'_f))\ \text{as required.}$$

Now, secondly, suppose $r \notin \pi'$. Then by Definition 4.4.4(page 128),

$$prune\ \pi'\ (t_1, r, t_2))) = (prune\ (\pi'_t \cup \{r\}, \pi'_f)\ t_1, r, prune\ (\pi'_t, \pi'_f \cup \{r\})\ t_2)$$
$$=$$
$$(prune\ \pi'\ t_1, r, prune\ \pi'\ t_2)\ \text{since } (t_1, r, t_2) \text{ is simple.}$$

So

$$pfun\ (prune\ \pi'\ (t_1, r, t_2)) = addleft(r, pfun\ (prune\ \pi'\ t_1)) \cup addright(r, pfun\ prune\ \pi' t_2))$$

So, since we are assuming that $r \in \pi_t$,

$$pfun\ (prune\ \pi'\ (t_1, r, t_2))\pi = addleft(r, pfun\ (prune\ \pi'\ t_1))\pi = pfun\ (prune\ \pi'\ t_1)\pi''.$$

Clearly, $diffs(\pi'', \pi') = \emptyset$ so by induction hypothesis, since $\pi'' \in dom(pfun\ t_1)$),

$$(\pi''_t - \pi'_t, \pi''_f - \pi'_f) \in dom(pfun\ (prune\ \pi'\ t_1))$$
$$\text{and}$$
$$pfun\ t_1\ \pi'' = pfun\ (prune\ \pi'\ t_1)\ (\pi''_t - \pi'_t, \pi''_f - \pi'_f).$$

But

$$(\pi_t - \pi'_t, \pi_f - \pi'_f) = (\pi''_t \cup \{r\} - \pi'_t, \pi''_f - \pi'_f) \in dom\ addleft(r, pfun\ (prune\ \pi'\ t_1))$$

and

$$dom\ addleft(r, pfun\ (prune\ \pi'\ t_1)) \subseteq dom\ pfun\ (prune\ \pi'\ (t_1, r, t_2))$$

so

$$(\pi_t - \pi'_t, \pi_f - \pi'_f) \in dom\ pfun\ (prune\ \pi'\ (t_1, r, t_2)) \text{ as required.}$$

Also,

$$pfun(t_1, r, t_2)\pi = addleft(r, pfun\ t_1)\pi \text{ (since } r \in \pi_t).$$
$$=$$
$$(pfun\ t_1)\pi'' \text{ (by definition of } addleft)$$
$$=$$
$$pfun\ (prune\ \pi'\ t_1)\ (\pi''_t - \pi'_t, \pi''_f - \pi'_f) \text{ by induction hypothesis(see above).}$$
$$=$$
$$pfun\ (prune\ \pi'\ (t_1, r, t_2))\ (\pi_t - \pi'_t, \pi_f - \pi'_f) \text{ by definiton of } prune$$
$$\text{(Definition 4.4.4(page 128)), since } r \in \pi_t - \pi'_t \text{, as required.}$$

In this part of the proof, we assumed first that $\pi \in dom(addleft(r, pfun\ t_1))$. We should now prove the whole theorem again, this time with the assumption that $\pi \in dom(addright(r, pfun\ t_2))$. An appeal to symmetry, however, allows this part of the proof to be omitted.

5.3.6 'Disagreement' Lemma

The next lemma states two distinct paths of the same symbolic execution tree must 'disagree' somewhere. In other words, their differences are not disjoint.

Lemma 5.3.5 *Let t be a simple symbolic execution tree then for all paths π, π' in $dom(pfun\ t)$),*

$$diffs(\pi, \pi') = \emptyset \iff \pi = \pi'$$

Proof: Induction on the depth of t (very straightforward so omitted).

5.3.7 'No Subpaths' Lemma

The next lemma states that in a simple symbolic execution tree there do not exist distinct paths π and π' with $\pi' \sqsubseteq \pi$.

Corollary 5.3.1 *Let t be a simple symbolic execution tree. Then*

$$\pi' \sqsubseteq \pi \text{ and } \pi' \in dom(pfun\ t) \text{ and } \pi \in dom(pfun\ t) \implies \pi = \pi'$$

Proof:

By Lemma 5.3.3(page 147), $\pi' \sqsubseteq \pi \implies diffs(\pi, \pi') = \emptyset$, which, by Lemma 5.3.5(page 151), implies that $\pi' = \pi$.

Lemma 5.3.6 *Let t be a simple symbolic execution tree and π a path, then for all π' in $dom(pfun(prune\ \pi\ t))$,*

$$diffs(\pi, \pi') = \emptyset$$

Proof: Induction on the depth of t base case trivial.

Let (t_1, r, t_2) be a simple. Let π be path, and let π' in $dom(pfun(prune\ \pi\ (t_1, r, t_2)))$.

case1 $r \notin \pi$

$$pfun\ prune\ \pi\ (t_1, r, t_2))) = addleft(r, pfun\ (prune\ \pi\ t_1)) \cup addright(r, pfun\ prune\ \pi\ t_2))$$

so

$$dom(pfun(prune\ \pi\ (t_1, r, t_2))) = \bigcup_{(\pi''_t, \pi''_f) \in dom\ pfun\ (prune\ \pi\ t_1) \cup dom\ pfun\ (prune\ \pi\ t_2)} \{(\pi''_t \cup \{r\}, \pi''_f)\} \cup \{(\pi''_t, \pi''_f \cup \{r\})\}.$$

By induction hypothesis,

$$diffs(\pi'', \pi) = \emptyset$$

Since $r \notin \pi$, $diffs((\pi''_t \cup \{r\}, \pi''_f), \pi) = \emptyset$ and $diffs((\pi''_t, \pi''_f) \cup \{r\}, \pi) = \emptyset$.
So for all $\pi' \in dom(pfun(prune\ \pi\ (t_1, r, t_2)))$,

$$diffs(\pi, \pi') = \emptyset$$

as required.

case2 $r \in \pi_t$

by Definition 4.4.2(page 125), $pfun\ prune\ \pi\ (t_1, r, t_2))) = pfun\ (prune\ \pi\ t_1))$

so result follows immediately by induction hypothesis.

case3 $r \in \pi_f$ similarly.

5.3.8 Joining Paths

An operator $\sqcup$ for 'joining' paths is now defined.

Definition 5.3.2

let π and π' be paths.

$$\pi \sqcup \pi' = (\pi_t \cup \pi'_t, \pi_f \cup \pi'_f)$$

Lemma 5.3.7 *$diffs(\pi, \pi') = \emptyset \iff \pi \sqcup \pi'$ is a path.*

Proof: trivial.

5.3.9 'Correspondence' Lemma

Lemma 5.3.8

- *Let σ be a state.*
- *Let s be a loop–free schema.*
- *Let ψ be a symbolic state obtained from s.*
- *Let δ be a symbolic value obtained from s.*
- *Let $p \in [s]$.*
- *Let σ' be the state derived from ψ with respect to (s, p, σ)*

Then the value derived from δ with respect to (s, p, σ') is equal to the value derived from evaldelta ψ δ with respect to (s, p, σ). i.e.

$$evalsym\ s\ p\ \sigma'\ \delta = evalsym\ s\ p\ \sigma\ (evaldelta\ \psi\ \delta)$$

Proof:

Induction on the depth of δ.

base case

δ is a variable.

L.H.S.=

$$evalsym\ s\ p\ (\lambda\ z.evalsym\ s\ p\ \sigma\ (\psi\ z))\ \delta$$

$$=$$

$$evalsym\ s\ p\ \sigma\ (\psi\ \delta)(\text{ by Definition 5.2.1(page 141))}$$

$$=$$

$$evalsym\ s\ p\ \sigma\ (evaldelta\ \psi\ \delta)\ \text{(by Definition 4.4.6(page 129))}$$

= R.H.S. as required.

If δ is of the form $f(S)$, then

L.H.S.=

$$evalsym\ s\ p\ (\lambda\ z.evalsym\ s\ p\ \sigma\ (\psi\ z))\ f(S)$$

$$=$$

$$\mathcal{E}[\![p\ f]\!]\bigcup_{\delta\in S} varof(s,\delta) \mapsto (evalsym\ s\ p\ (\lambda\ z.evalsym\ s\ p\ \sigma\ (\psi\ z))\ \delta)\ \text{(by Definition 5.2.1(page 141))}$$

$$=$$

$$\mathcal{E}[\![p\ f]\!]\bigcup_{\delta\in S} varof(s,\delta) \mapsto (evalsym\ s\ p\ \sigma\ (evaldelta\ \psi\ \delta))\ \text{by induction hypothesis.}$$

R.H.S.=

$$evalsym\ s\ p\ \sigma\ (evaldelta\ \psi\ f(S))$$

$$=$$

$$evalsym\ s\ p\ \sigma\ (f(\bigcup_{\delta\in S} evaldelta\ \psi\ \delta))\ \text{(by Definition 4.4.6(page 129))}$$

$$=$$

$$\mathcal{E}[\![p\ f]\!]\bigcup_{\delta\in S} varof(s,\delta) \mapsto (evalsym\ s\ p\ \sigma\ (evaldelta\ \psi\ \delta))\ \text{(by Definition 5.2.1(page 141))}$$

=L.H.S as required.

5.3.10 Evaluating a Path in a Symbolic State

To evaluate a path π in a symbolic state ψ we simply evaluate each symbolic predicate of π in ψ.

Definition 5.3.3 ($pathinstate : \Psi \rightarrow path \rightarrow path$)

$$pathinstate\ \psi\ (\pi_1,\pi_2) = (\bigcup_{\delta\in\pi_1} evaldelta\ \psi\ \delta,\ \bigcup_{\delta\in\pi_2} evaldelta\ \psi\ \delta)$$

Lemma 5.3.9 *Let t_1 and t_2 be simple symbolic execution trees and let $\pi_1 \in dom\ (pfun\ t_1)$ and $\pi_2 \in dom\ (pfun\ t_2)$ be paths such that*

$$diffs(\pi_1, pathinstate\ (pfun\ t_1\pi_1)\ \pi_2) = \emptyset$$

then for all variables z,

$$evaldelta\ (pfun\ t_1\ \pi_1)\ (pfun\ t_2\ \pi_2\ z)$$
$$=$$
$$pfun\ (treeinstate\ (pfun\ t_1\ \pi_1)\ t_2)\ (pathinstate\ (pfun\ t_1\pi_1)\ \pi_2)\ z.$$

Proof: Induction on the depth of t_2.

<u>Base Case</u> t_2 is a leaf ψ

in which case π_2 must be $(\emptyset, \emptyset)$. Therefore

$$\text{LHS}=$$
$$evaldelta\ (pfun\ t_1\ \pi_1)\ (pfun\ t_2\ \pi_2\ z)$$
$$=$$
$$evaldelta\ (pfun\ t_1\ \pi_1)\ (\psi\ z)\ \text{since } t_2 \text{ is a leaf } \psi$$

and

$$\text{RHS}=$$
$$pfun\ (treeinstate\ (pfun\ t_1\ \pi_1)\ t_2)\ (pathinstate\ (pfun\ t_1\pi_1)\ \pi_2)\ z$$
$$=$$
$$pfun\ (treeinstate\ (pfun\ t_1\ \pi_1)\ \psi)(\emptyset, \emptyset)\ z (\text{by Definition 5.3.3(page 154)})$$
$$=$$
$$pfun\ (updatestateinstate\ (pfun\ t_1\ \pi_1)\ \psi)(\emptyset, \emptyset)\ z\ (\text{by Definition 4.4.8(page 130)})$$
$$=$$
$$updatestateinstate\ (pfun\ t_1\ \pi_1)\ \psi)\ z(\text{by Definition 4.4.2(page 125)})$$
$$=$$
$$evaldelta\ (pfun\ t_1\ \pi_1)\ (\psi\ z)\ (\text{by Definition 4.4.7(page 130)) as required.}$$

<u>Now let $t_2 = (t_L, r, t_R)$.</u>

$$\text{LHS}=$$
$$evaldelta\ (pfun\ t_1\ \pi_1)\ (pfun\ (t_L, r, t_R)\ \pi_2\ z)$$

$$=$$

$$evaldelta\ (pfun\ t_1\ \pi_1)\ (addleft(r, (pfun\ t_L)) \cup\ addright(r, pfun\ t_R))\ \pi_2\ z)$$

Suppose first that $\pi_2\ \in\ dom\ addleft(r, (pfun\ t_L))$ then

$$\text{LHS=}$$

$$evaldelta\ (pfun\ t_1\ \pi_1)\ (addleft(r, (pfun\ t_L))\ \pi_2\ z)$$

$$=$$

$$evaldelta\ (pfun\ t_1\ \pi_1)\ ((pfun\ t_L)\ (\pi_2 - (\{r\}, \emptyset))\ z).$$

Clearly,

$$diffs(\pi_1, pathinstate\ (pfun\ t_1\pi_1)\ (\pi_2 - (\{r\}, \emptyset))) = \emptyset.$$

So, by induction hypothesis,

$$\text{LHS=}$$

$$pfun\ (treeinstate\ (pfun\ t_1\ \pi_1)\ t_L)\ (pathinstate\ (pfun\ t_1\pi_1)\ (\pi_2 - (\{r\}, \emptyset))\ z$$

$$=$$

$$pfun\ (treeinstate\ (pfun\ t_1\ \pi_1)\ (t_L, r, t_R)\ (pathinstate\ (pfun\ t_1\pi_1)\ (\pi_2)\ z\ \text{(by Definition 4.4.2(page 125))}$$

as required.

The proof when $\pi_2\ \in\ dom\ addright(r, (pfun\ t_R))$ is symmetrical so omitted.

Lemma 5.3.10 *If t is simple so is treeinstate ψ t.*

Proof: trivial induction on the depth of t.

Lemma 5.3.11 *If t is simple,*

$$\pi \in dom\ (pfun\ t) \implies pathinstate\ \psi\ \pi\ \in dom\ (treeinstate\ \psi\ t).$$

Proof: trivial induction on the depth of t.

Lemma 5.3.12 *Let t_1 and t_2 be simple symbolic execution trees and let $\pi_1 \in dom\ (pfun\ t_1)$ and $\pi_2 \in dom\ (pfun\ t_2)$ be paths such that*

$$diffs(\pi_1, pathinstate\ (pfun\ t_1\pi_1)\ \pi_2) = \emptyset$$

then for all variables z,

$$\pi_1 \sqcup pathinstate\ (pfun\ t_1\pi_1)\ \pi_2 \in dom\ simplify\ (sequence\ t_1\ t_2)$$

and

$$pfun\ (treeinstate\ (pfun\ t_1\ \pi_1)\ t_2)\ (pathinstate\ (pfun\ t_1\pi_1)\ \pi_2)\ z =$$

$$pfun\ (simplify\ (sequence\ t_1\ t_2))(\pi_1 \sqcup pathinstate\ (pfun\ t_1\pi_1)\ \pi_2))z.$$

Proof: by induction on the depth of t_1.

<u>Base Case</u> t_1 is a leaf ψ.

First we must show

$$(\emptyset, \emptyset) \sqcup pathinstate\ (pfun\ \psi\ (\emptyset, \emptyset))\ \pi_2 \in dom\ simplify\ (sequence\ \psi\ t_2).$$

i.e.

$$pathinstate\ \psi\ \pi_2 \in dom\ simplify\ (sequence\ \psi\ t_2).$$

i.e.

$$pathinstate\ \psi\ \pi_2 \in dom\ simplify\ (treeinstate\ \psi\ t_2)$$

which follows immediately from Lemma 5.3.10(page 156) and Lemma 5.3.11(page 156).

We must now show

$$pfun\ (treeinstate\ (\psi\ t_2)\ (pathinstate\ (\psi\ \pi_2)\ z =$$

$$pfun\ (simplify\ (sequence\ \psi\ t_2))(pathinstate\ (\psi\ \pi_2))z.$$

RHS=

$$pfun\ (simplify\ (sequence\ \psi\ t_2))(pathinstate\ (\psi\ \pi_2))z$$

$$=$$

$$pfun\ (simplify\ (treeinstate\ \psi\ t_2))(pathinstate\ (\psi\ \pi_2))z \text{ (by Definition 4.4.9(page 131))}$$

$$=$$

$$pfun\ (treeinstate\ \psi\ t_2)(pathinstate\ (\psi\ \pi_2))z \text{ (by Lemma 5.3.10(page 156))=LHS.}$$

This completes the proof of the base case.

<u>Now let $t_1 = (t_L, r, t_R)$.</u>

Let $\pi_1 \in dom\ (pfun\ (t_L, r, t_R))$ and $\pi_2 \in dom\ (pfun\ t_2)$ be paths such that

$$diffs(\pi_1, pathinstate\ (pfun\ (t_L, r, t_R)\pi_1)\ \pi_2) = \emptyset.$$

We must show that

1. $\pi_1 \sqcup pathinstate\ (pfun\ (t_L, r, t_R)\pi_1)\ \pi_2 \in dom\ simplify\ (sequence\ (t_L, r, t_R)\ t_2)$

   and

2. for all variables z,

$$pfun\ (treeinstate\ (pfun\ (t_L, r, t_R)\ \pi_1)\ t_2)\ (pathinstate\ (pfun\ (t_L, r, t_R)\pi_1)\ \pi_2)\ z$$
$$=$$
$$pfun\ (simplify\ (sequence\ (t_L, r, t_R)\ t_2))(\pi_1 \sqcup pathinstate\ (pfun\ (t_L, r, t_R)\pi_1)\ \pi_2))z.$$

As in previous proofs, suppose first that

$$\pi_1 \in dom\ (addleft(r, pfun\ t_L)).$$

Then

$$\pi_1 - (\{r\}, \emptyset) \in dom\ (pfun\ t_L).$$

So by induction hypothesis,

$$\pi_1 - (\{r\}, \emptyset) \sqcup pathinstate\ (pfun\ t_L(\pi_1 - (\{r\}, \emptyset))\ \pi_2) \in dom\ simplify\ (sequence\ t_L\ t_2).$$

Therefore

1. $\pi_1 \sqcup pathinstate\ (pfun\ (t_L, r, t_R)\ \pi_1)\ \pi_2 \in dom\ simplify\ (sequence\ (t_L, r, t_R)\ t_2)$

   as required and

2. by induction hypothesis, for all variables z,

$$pfun\ (treeinstate\ (pfun\ t_L\ (\pi_1 - (\{r\}, \emptyset))\ t_2)\ (pathinstate\ (pfun\ t_L\ (\pi_1 - (\{r\}, \emptyset))\ \pi_2)\ z$$
$$=$$
$$pfun\ (simplify\ (sequence\ t_L\ t_2))((\pi_1 - (\{r\}, \emptyset) \sqcup pathinstate\ (pfun\ t_L\ (\pi_1 - (\{r\}, \emptyset))\ \pi_2))z.$$

   Therefore for all variables z,

$$pfun\ (treeinstate\ (pfun\ (t_L, r, t_R)\ \pi_1)\ t_2)\ (pathinstate\ (pfun\ (t_L, r, t_R)\pi_1)\ \pi_2)\ z$$

$$=$$

$$pfun\ (treeinstate\ (pfun\ t_L\ )\ (\pi_1 - (\{r\}, \emptyset))\ t_2)\ (pathinstate\ (pfun\ t_L\ )\ (\pi_1 - (\{r\}, \emptyset)\ \pi_2)\ z$$

(by Definition 4.4.2(page 125))

$$=$$

$$pfun\ (simplify\ (sequence\ t_L\ t_2))((\pi_1 - (\{r\}, \emptyset) \sqcup pathinstate\ (pfun\ t_L\ (\pi_1 - (\{r\}, \emptyset))\ \pi_2))z$$

(by induction hypothesis)

$$=$$

$$pfun\ (simplify\ (sequence\ (t_L, r, t_R)\ t_2))(\pi_1 \sqcup pathinstate\ (pfun\ (t_L, r, t_R)\pi_1)\ \pi_2))z$$ (by Definition 4.4.2(page 125))

as required. Again, we appeal to symmetry in order to justify the omission of the case where $\pi_1 \in dom\ (addright(r, pfun\ t_R))$.

5.3.11 The 'One Path' Lemma

Lemma 5.3.13 *Given a schema s, a program $p \in [s]$ and a state σ, and a simple symbolic execution tree, t, all of whose non–leaf nodes are obtained from s, then there is exactly one path π of t such that satisfy $t\ p\ \sigma\ \pi$.*

Proof: : trivial.

5.4 The Soundness and Completeness of $\mathcal{S}$

We are now in a position to prove the main theorem of this chapter: that $\mathcal{S}$ is sound and complete.

1. Complete, in the sense that for all loop free schemas s, for all programs p in $[s]$, and for all states σ, there is exactly one path in the symbolic execution tree of s that is 'satisfied' in state σ with respect to p and the state at the end of this path 'corresponds' to $\mathcal{M}[\![p]\!]\sigma$ (i.e. the normal denotational meaning of p in σ).

   and

2. sound, in the sense that for all paths π of the symbolic execution tree of s there is a p in $[s]$ and a state σ such that π is 'satisfied' in state σ with respect to p and the state

at the end of the path again corresponds to $\mathcal{M}[\![p]\!]\sigma$.

Theorem 5.4.1 (Soundness and Completeness of $\mathcal{S}$) *Given a loop-free schema s, and a program $p \in [s]$, and a state, σ, there exists exactly one path $\pi \in dom\ (pfun\ \mathcal{S}[\![s]\!])$ such that satisfy $s\ p\ \sigma\ \pi$ and in this case, for all z,*

$$\mathcal{M}[\![p]\!]\sigma\ z = evalsym\ s\ p\ \sigma\ (pfun\ \mathcal{S}[\![s]\!]\ \pi\ z).$$

Conversely, for all $\pi \in dom\ (pfun\ \mathcal{S}[\![s]\!])$ there exists a program $p \in [s]$, and a state, σ such that

$$satisfy\ s\ p\ \sigma\ \pi$$
$$and$$
$$for\ all\ z,\ \mathcal{M}[\![p]\!]\sigma\ z = evalsym\ s\ p\ \sigma\ (pfun\ \mathcal{S}[\![s]\!]\ \pi\ z).$$

Proof: by induction on the structure of s.

1. s is *skip*.

   By Definition 4.5.3(page 132), $\mathcal{S}[\![skip]\!]$ is the symbolic execution tree consisting of the single leaf node $\lambda v.v$. Therefore

   $$pfun\ \mathcal{S}[\![skip]\!] = \{(\emptyset, \emptyset) \mapsto \lambda v.v\}$$

   for all z,

   $$\text{LHS}{=}\mathcal{M}[\![skip]\!]\sigma\ z$$
   $$=$$
   $$\sigma\ z \text{ (by definition of } \mathcal{M})$$
   $$=$$
   $$evalsym\ skip\ skip\ \sigma\ z \text{ (by Definition 5.2.1(page 141))}$$
   $$=$$
   $$evalsym\ skip\ skip\ \sigma\ (pfun\ \mathcal{S}[\![skip]\!]\ (\emptyset, \emptyset)\ z)\text{(by Definition 4.5.3(page 132))}$$
   $$=\text{RHS.}$$

2. s is *FAIL*.

   By Definition 4.5.2(page 132), $\mathcal{S}[\![FAIL]\!]$ is the symbolic execution tree consisting of the single leaf node $\bot$. Therefore

$$pfun\ \mathcal{S}[\![FAIL]\!] = \{(\emptyset,\emptyset) \mapsto \bot\}.$$

For all z,

$$\begin{gathered}\text{LHS}{=}\mathcal{M}[\![FAIL]\!]\sigma\ z\\ =\\ \bot\ \text{(by definition of } \mathcal{M}\text{)}\\ =\\ evalsym\ FAIL\ FAIL\ \sigma\ \bot\ \text{(by Definition 5.2.1(page 141))}\\ =\\ evalsym\ FAIL\ FAIL\ \sigma\ (pfun\ \mathcal{S}[\![FAIL]\!]\ (\emptyset,\emptyset)\ z)\text{(by Definition 4.5.2(page 132))}\\ {=}\text{RHS.}\end{gathered}$$

3. s is a single assignment statement $x := f(V)$.

   Then for all p in $[s]$, p is a single assignment of the form $x := E$ where E is an expression with $ref\ E = V$.

$$\mathcal{M}[\![p]\!]\sigma\ z = \begin{cases} \sigma\ z & \text{if } z \neq x \\ \mathcal{E}[\![E]\!]\sigma & \text{if } z = x \end{cases}$$

By Definition 4.5.1(page 131), $\mathcal{S}[\![x{:=}f(V)]\!]$ is the Symbolic Execution Tree consisting of the single leaf state:-

$$\mathcal{S}[\![x{:=}f(V)]\!] = \lambda z. \begin{cases} z & \text{if } z \neq x \\ f(V) & \text{if } z = x \end{cases}$$

So, by Definition 4.4.2(page 125),

$$pfun\ \mathcal{S}[\![s]\!] = \{(\emptyset,\emptyset) \mapsto \mathcal{S}[\![x{:=}f(V)]\!]\}.$$

Therefore

$$evalsym\ s\ p\ \sigma\ (pfun\ \mathcal{S}[\![s]\!]\ (\emptyset,\emptyset)\ z)$$

$$=$$

$$evalsym\ s\ p\ \sigma\ (\mathcal{S}[\![x{:=}f(V)]\!]\ z)$$

$$= \begin{cases} evalsym\ s\ p\ \sigma\ z & \text{if } z \neq x \\ evalsym\ s\ p\ \sigma\ f(V) & \text{if } z = x \end{cases} \quad \text{(by Definition 4.5.1(page 131))}$$

$$= \begin{cases} \sigma\ z & \text{if } z \neq x \\ \mathcal{E}[\![E]\!] \bigcup_{\delta \in V} varof(s,\delta) \mapsto (evalsym\ s\ p\ \sigma\ \delta) & \text{if } z = x \end{cases} \quad \text{(by Definition 5.2.1(page 141))}$$

$$= \begin{cases} \sigma\ z & \text{if } z \neq x \\ \mathcal{E}[\![E]\!] \bigcup_{v \in V} v \mapsto \sigma\ v & \text{if } z = x \end{cases} \quad \text{(by Definition 5.2.1(page 141))}$$

which by Lemma 3.4.1(page 89) gives

$$\begin{cases} \sigma\ z & \text{if } z \neq x \\ \mathcal{E}[\![E]\!]\sigma & \text{if } z = x \end{cases}$$

$= \mathcal{M}[\![p]\!]\sigma\ z$ as required.

4. s is the conditional statement schema: *if* $f(V)$ *then* s_1 *else* s_2.

   Assume theorem is true for schemas s_1 and s_2.

   Let $p \in [\textit{if } f(V) \textit{ then } s_1 \textit{ else } s_2]$ and let σ be a state.

   Since $p \in [\textit{if } f(V) \textit{ then } s_1 \textit{ else } s_2]$, p is of the form $[\![\textit{if } E \textit{ then } p_1 \textit{ else } p_2]\!]$, where

   (a) $ref\ E = V$,

   (b) $p_1 \in [s_1]$ and

   (c) $p_2 \in [s_2]$.

   We must show that there exists exactly one path $\pi \in dom\ (pfun\ \mathcal{S}[\![\textit{if } f(V) \textit{ then } s_1 \textit{ else } s_2]\!])$ such that $satisfy\ [\![\textit{if } f(V) \textit{ then } s_1 \textit{ else } s_2]\!]\ [\![\textit{if } E \textit{ then } p_1 \textit{ else } p_2]\!]\ \sigma\ \pi$ and in this case, for all z,

$$\mathcal{M}[\![\textit{if } E \textit{ then } p_1 \textit{ else } p_2]\!]\sigma\ z$$

$$=$$

$$evalsym\ [\![\textit{if } f(V) \textit{ then } s_1 \textit{ else } s_2]\!]\ [\![\textit{if } E \textit{ then } p_1 \textit{ else } p_2]\!]\ \sigma\ (pfun\ \mathcal{S}[\![\textit{if } f(V) \textit{ then } s_1 \textit{ else } s_2]\!]\ \pi\ z).$$

   By induction hypothesis, for all programs $p \in [s_1]$ and for all states σ there exists exactly one path $\pi \in dom\ (pfun\ \mathcal{S}[\![s_1]\!])$ such that *satisfy* $s\ p\ \sigma\ \pi$ and in this case, for all z,

$$\mathcal{M}[\![p]\!]\sigma\ z = evalsym\ s_1\ p\ \sigma\ (pfun\ \mathcal{S}[\![s_1]\!]\ \pi\ z).$$

Similarly, for all programs $p \in [s_2]$ and for all states σ there exists exactly one path $\pi \in dom\ (pfun\ \mathcal{S}[\![s_2]\!])$ such that *satisfy* $s_2\ p\ \sigma\ \pi$ and in this case, for all z,

$$\mathcal{M}[\![p]\!]\sigma\ z = evalsym\ s\ p\ \sigma\ (pfun\ \mathcal{S}[\![s_2]\!]\ \pi\ z).$$

Let σ be a state for which $[\![if\ E\ then\ p_1\ else\ p_2]\!]$ is defined. If $\mathcal{E}[\![E]\!]\sigma = true$, then

$$\mathcal{M}[\![if\ E\ then\ p_1\ else\ p_2]\!]\sigma = \mathcal{M}[\![p_1]\!]\sigma.$$

By induction hypothesis therefore, there exists exactly one path $(\pi_t, \pi_f) \in dom\ (pfun\ \mathcal{S}[\![s_1]\!])$ such that *satisfy* $s\ p_1\ \sigma\ (\pi_t, \pi_f)$ and in this case, for all z,

$$\mathcal{M}[\![p_1]\!]\sigma\ z = evalsym\ s_1\ p_1\ \sigma\ (pfun\ \mathcal{S}[\![s_1]\!]\ (\pi_t, \pi_f)\ z).$$

Clearly, $f(V) \notin \pi_f$ since $\mathcal{E}[\![E]\!]\sigma = true$,
so by Lemma 5.3.4(page 148), $(\pi_t - \{f(V)\}, \pi_f) \in dom\ (pfun\ prune(\{f(V)\}, \emptyset)\mathcal{S}[\![s_1]\!])$
but, by Definition 4.5.4(page 132),

$$pfun\ (\mathcal{S}[\![if\ f(V)\ then\ s_1\ else\ s_2]\!])$$
$$=$$
$$pfun(simplify\ (\mathcal{S}[\![s_1]\!], f(V), \mathcal{S}[\![s_2]\!]))$$
$$=$$
$$pfun(prune(\emptyset, \emptyset)\ (\mathcal{S}[\![s_1]\!], f(V), \mathcal{S}[\![s_2]\!]))\ \text{(by Definition 4.4.5(page 129))}$$
$$=$$
$$pfun(\ (prune(\{f(V)\}, \emptyset)\mathcal{S}[\![s_1]\!], f(V), prune(\emptyset, \{f(V)\})\mathcal{S}[\![s_2]\!]))$$
$$\text{(by Definition 4.4.4(page 128))}$$
$$=$$
$$addleft(f(V), pfun(\ (prune(\{f(V)\}, \emptyset)\mathcal{S}[\![s_1]\!]) \cup addright(f(V), prune(\emptyset, \{f(V)\})\mathcal{S}[\![s_2]\!])).$$
$$\text{(by Definition 4.4.2(page 125))}$$

$$\text{Therefore } (\pi_t \cup \{f(V)\}, \pi_f) \in dom(pfun\ (\mathcal{S}[\![if\ f(V)\ then\ s_1\ else\ s_2]\!])$$

and

$$satisfy\ [\![if\ f(V)\ then\ s_1\ else\ s_2]\!]\ [\![if\ E\ then\ p_1\ else\ p_2]\!]\ \sigma\ (\pi_t \cup \{f(V)\}, \pi_f).$$

By Definition 4.5.4(page 132),

$$evalsym\ [\![if\ f(V)\ then\ s_1\ else\ s_2]\!]\ [\![if\ E\ then\ p_1\ else\ p_2]\!]\ \sigma$$
$$(pfun\ \mathcal{S}[\![if\ f(V)\ then\ s_1\ else\ s_2]\!]\ (\pi_t \cup \{f(V)\}, \pi_f)\ z)$$
$$=$$
$$evalsym\ [\![if\ f(V)\ then\ s_1\ else\ s_2]\!]\ [\![if\ E\ then\ p_1\ else\ p_2]\!]\ \sigma$$
$$(pfun\ simplify\ (\mathcal{S}[\![s_1]\!], f(V), \mathcal{S}[\![s_2]\!])\ (\pi_t \cup \{f(V)\}, \pi_f)\ z)$$
$$=$$
$$evalsym\ [\![if\ f(V)\ then\ s_1\ else\ s_2]\!]\ [\![if\ E\ then\ p_1\ else\ p_2]\!]\ \sigma$$
$$(pfun\ (prune(f(V), \emptyset)\mathcal{S}[\![s_1]\!], f(V), prune(\emptyset, f(V))\mathcal{S}[\![s_2]\!])\ (\pi_t \cup \{f(V)\}, \pi_f)\ z)$$
$$\text{(by Definition 4.4.5(page 129))}$$
$$=$$
$$evalsym\ [\![if\ f(V)\ then\ s_1\ else\ s_2]\!]\ [\![if\ E\ then\ p_1\ else\ p_2]\!]\ \sigma$$
$$(pfun\ prune(f(V), \emptyset)\mathcal{S}[\![s_1]\!]\ (\pi_t - \{f(V)\}, \pi_f)\ z)$$
$$\text{(by Definition 4.4.2(page 125))}$$
$$=$$
$$evalsym\ [\![if\ f(V)\ then\ s_1\ else\ s_2]\!]\ [\![if\ E\ then\ p_1\ else\ p_2]\!]\ \sigma\ (pfun\ \mathcal{S}[\![s_1]\!]\ (\pi_t, \pi_f)\ z)$$
$$\text{(by Lemma 5.3.4(page 148), using the fact that } f(V) \notin \pi_f)$$
$$=$$
$$evalsym\ [\![s_1]\!]\ [\![p_2]\!]\ \sigma\ (pfun\ \mathcal{S}[\![s_1]\!]\ (\pi_t, \pi_f)\ z)$$
$$=$$
$$\mathcal{M}[\![p_1]\!]\sigma \text{(by induction hypothesis)}$$
$$=$$
$$\mathcal{M}[\![if\ E\ then\ p_1\ else\ p_2]\!]\sigma \text{ since } \mathcal{E}[\![E]\!]\sigma = true \text{ as required.}$$

Similarly, if $\mathcal{E}[\![E]\!]\sigma = false$ (by symmetry).

We now prove the converse, namely,

for all $\pi \in dom\ (pfun\ \mathcal{S}[\![if\ f(V)\ then\ s_1\ else\ s_2]\!])$

there exists a program $p = [\![if\ E\ then\ p_1\ else\ p_2]\!] \in [if\ f(V)\ then\ s_1\ else\ s_2]$,

and a state, σ such that

$$satisfy\ [\![if\ f(V)\ then\ s_1\ else\ \ s_2]\!]\ [\![if\ E\ then\ p_1\ else\ p_2]\!]\ \sigma\ \pi$$

and

for all z,

$$\mathcal{M}[\![if\ E\ then\ p_1\ else\ p_2]\!]\sigma\ z$$

$$=$$

$$evalsym\ [\![if\ f(V)\ then\ s_1\ else\ \ s_2]\!]\ [\![if\ E\ then\ p_1\ else\ p_2]\!]\ \sigma\ (pfun\ \mathcal{S}[\![if\ f(V)\ then\ s_1\ else\ \ s_2]\!]\ \pi\ z).$$

Proof:

Let $\pi \in dom\ (pfun\ \mathcal{S}[\![if\ f(V)\ then\ s_1\ else\ s_2]\!])$

then by Definition 4.5.4(page 132),

$$\pi \in dom\ (pfun(simplify(\mathcal{S}[\![s_1]\!], f(V), \mathcal{S}[\![s_2]\!]))$$

$$=$$

$$dom\ (pfun(prune(\emptyset,\emptyset)(\mathcal{S}[\![s_1]\!], f(V), \mathcal{S}[\![s_2]\!]))\ \text{(by Definition 4.4.5(page 129))}$$

$$=$$

$$dom\ (pfun(prune(\{f(V)\},\emptyset)\mathcal{S}[\![s_1]\!], f(V), prune(\emptyset,\{f(V)\})\mathcal{S}[\![s_2]\!]))$$

(Definition 4.4.4(page 128))

$$=$$

$$dom\ (addleft(f(V), pfun(prune(\{f(V)\},\emptyset)\mathcal{S}[\![s_1]\!]) \cup$$
$$addright(f(V), pfun(prune(\emptyset,\{f(V)\})\mathcal{S}[\![s_2]\!])).\ \text{(by Definition 4.4.2(page 125))}$$

Suppose first, that

$$\underline{\pi \in dom\ (addleft(f(V), pfun(prune(\{f(V)\},\emptyset)\mathcal{S}[\![s_1]\!]))}$$

Case1 $f(V)$ is not a node of $\mathcal{S}[\![s_1]\!]$

In which case

$$\pi \in dom\ (addleft(f(V), pfun(\mathcal{S}[\![s_1]\!])).$$

So $\pi = (\pi'_t \cup \{f(V)\}, \pi' f)$ for some path $\pi' \in dom(pfun(\mathcal{S}[\![s_1]\!]))$ that does not contain $f(V)$. By induction hypothesis, therefore, there exists a program $p_1 \in [s_1]$, and a state, σ such that

$$satisfy\ s_1\ p_1\ \sigma\ \pi'$$
$$\text{and}$$
$$\text{for all } z,\ \mathcal{M}[\![p_1]\!]\sigma\ z = evalsym\ s_1\ p_1\ \sigma\ (pfun\ \mathcal{S}[\![s_1]\!]\ \pi'\ z).$$

By Assumption 3.4.1, there is an expression E which references the variables V such that $\mathcal{M}[\![E]\!]\sigma = true$. Therefore

$$satisfy\ [\![if\ f(V)\ then\ s_1\ else\ \ s_2]\!]\ [\![if\ E\ then\ p_1\ else\ p_2]\!]\ \sigma\ \pi$$
$$\text{and}$$
$$\text{for all } z,\ \mathcal{M}[\![if\ E\ then\ p_1\ else\ p_2]\!]\sigma\ z =$$
$$evalsym\ [\![if\ f(V)\ then\ s_1\ else\ \ s_2]\!]\ [\![if\ E\ then\ p_1\ else\ p_2]\!]\ \sigma\ (pfun\ \mathcal{S}[\![if\ f(V)\ then\ s_1\ else\ \ s_2]\!]\ \pi\ z).$$

<u>Case</u>2 $f(V)$ is a node of $\mathcal{S}[\![s_1]\!]$.

By induction hypothesis and Lemma 5.3.2(page 145),

for all paths π' in $pfun(prune(\{f(V)\}, \emptyset)\mathcal{S}[\![s_1]\!]))$ there exists a program $p_1 \in [s_1]$, and a state, σ such that

$$satisfy\ s_1\ p_1\ \sigma\ \pi'$$
$$\text{and}$$
$$\text{for all } z,\ \mathcal{M}[\![p_1]\!]\sigma\ z = evalsym\ s_1\ p_1\ \sigma\ (pfun\ (prune(\{f(V)\}, \emptyset)\mathcal{S}[\![s_1]\!]\ \pi'\ z).$$

Again, by Assumption 3.4.1, choose an expression E which references the variables V such that $\mathcal{M}[\![E]\!]\sigma = true$ and the result follows immediately.

Exactly as in Lemma 5.3.4(page 148), we appeal to symmetry in order to allow ourselves the luxury of omitting the proof of the case when

$$\pi \in dom\ (addright(f(V), pfun(prune(\{f(V)\}, \emptyset)\mathcal{S}[\![s_2]\!])).$$

This completes the proof for conditionals.

5. Let s be the sequence $[\![s_1; s_2]\!]$. We first must prove that given a program $p_1; p_2$ in $[s_1; s_2]$, and a state, σ, there exists exactly one path $\pi \in dom\ (pfun\ \mathcal{S}[\![s_1; s_2]\!])$ such that

satisfy $[\![s_1; s_2]\!]$ $[\![p_1, p_2]\!]$ σ π and in this case, for all z,

$$\mathcal{M}[\![p_1;p_2]\!]\sigma\ z = \mathit{evalsym}\ [\![s_1;s_2]\!]\ [\![p_1;p_2]\!]\ \sigma\ (\mathit{pfun}\ \mathcal{S}[\![s_1;s_2]\!]\ \pi\ z).$$

i.e.by definition of $\mathcal{M}$ and Definition 4.5.5(page 132), we must show that there exists a path π in $\mathit{dom}(\mathit{pfun}(\mathit{simplify}(\mathit{sequence}\ \mathcal{S}[\![s_1]\!]\ \mathcal{S}[\![s_2]\!])))$ such that

$$\mathit{satisfy}\ [\![s_1;s_2]\!]\ [\![p_1,p_2]\!]\ \sigma\ \pi$$

and in this case, for all z

$$\mathcal{M}[\![p_2]\!](\mathcal{M}[\![p_1]\!]\sigma)z = \mathit{evalsym}\ [\![s_1;s_2]\!]\ [\![p_1;p_2]\!]\ \sigma\ ((\mathit{pfun}(\mathit{simplify}(\mathit{sequence}\ \mathcal{S}[\![s_1]\!]\ \mathcal{S}[\![s_2]\!])))\pi\ z).$$

Proof: By induction hypothesis, there exists exactly one path π_1 in $\mathit{dom}\ (\mathit{pfun}\ \mathcal{S}[\![s_1]\!])$ such that *satisfy* $[\![s_1]\!]$ $[\![p_1]\!]$ σ π_1 and in this case, for all z,

$$\mathcal{M}[\![p_1]\!]\sigma\ z = \mathit{evalsym}\ [\![s_1]\!]\ [\![p_1]\!]\ \sigma\ (\mathit{pfun}\ \mathcal{S}[\![s_1]\!]\ \pi_1\ z).$$

and by induction hypothesis there exists exactly one path π_2 in $\mathit{dom}\ (\mathit{pfun}\ \mathcal{S}[\![s_2]\!])$ such that

$$\mathit{satisfy}\ [\![s_2]\!]\ [\![p_2]\!]\ (\mathcal{M}[\![p_1]\!]\sigma)\ \pi_2$$

and for all z,

$$\mathcal{M}[\![p_2]\!](\mathcal{M}[\![p_1]\!]\sigma)z = \mathit{evalsym}\ [\![s_2]\!]\ [\![p_2]\!]\ (\mathcal{M}[\![p_1]\!]\sigma)\ ((\mathit{pfun}(\mathcal{S}[\![s_2]\!]))\pi_2\ z).$$

Therefore there exists a path π_1 in $\mathit{dom}\ (\mathit{pfun}\ \mathcal{S}[\![s_1]\!])$ such that

$$\mathcal{M}[\![p_2]\!](\mathcal{M}[\![p_1]\!]\sigma)z$$
$$=$$
$$\mathit{evalsym}\ [\![s_2]\!]\ [\![p_2]\!]\ (\lambda x.\mathit{evalsym}\ [\![s_1]\!]\ [\![p_1]\!]\ \sigma\ (\mathit{pfun}\ \mathcal{S}[\![s_1]\!]\ \pi_1\ x))\ ((\mathit{pfun}(\mathcal{S}[\![s_2]\!]))\pi_2\ z).$$

Therefore

$$\mathcal{M}[\![p_2]\!](\mathcal{M}[\![p_1]\!]\sigma)z$$
$$=$$

$$evalsym\ [\![s_1; s_2]\!]\ [\![p_1; p_2]\!]\ (\lambda x.evalsym\ [\![s_1]\!]\ [\![p_1]\!]\ \sigma\ (pfun\ \mathcal{S}[\![s_1]\!]\ \pi_1\ x))\ ((pfun(\mathcal{S}[\![s_2]\!]))\pi_2\ z)$$
(since $[\![s_1, s_2]\!]$ is a valid schema and $[\![p_1, p_2]\!]$ is a valid program)
$$=$$
$$evalsym\ [\![s_1; s_2]\!]\ [\![p_1; p_2]\!]\ \sigma\ evaldelta((pfun\ \mathcal{S}[\![s_1]\!])\ \pi_1))\ ((pfun(\mathcal{S}[\![s_2]\!]))\pi_2\ z$$
(by Lemma 5.3.8(page 153))

By Lemma 5.3.8(page 153)

$$diffs(\pi_1, pathinstate(pfun\ \mathcal{S}[\![s_1]\!])\ \pi_1)\ \pi_2) = \emptyset.$$

Therefore, by Lemma 5.3.9(page 155),

$$\mathcal{M}[\![p_2]\!](\mathcal{M}[\![p_1]\!]\sigma)z$$
$$=$$
$$evalsym\ [\![s_1; s_2]\!]\ [\![p_1; p_2]\!]\ \sigma\ (pfun\ (treeinstate\ (pfun\ \mathcal{S}[\![s_1]\!]\ \pi_1))\ (\mathcal{S}[\![s_2]\!]))\ (pathinstate(pfun\ \mathcal{S}[\![s_1]\!]\ \pi_1)\ \pi_2)\ z$$

But by Lemma 5.3.12(page 156),

$$\pi_1 \sqcup (pathinstate(pfun\ \mathcal{S}[\![s_1]\!]\ \pi_1)\ \pi_2) \in dom\ simplify\ (sequence\ \mathcal{S}[\![s_1]\!]\ \mathcal{S}[\![s_2]\!])$$
and
$$(pfun\ (treeinstate\ (pfun\ \mathcal{S}[\![s_1]\!]\ \pi_1))\ (\mathcal{S}[\![s_2]\!]))\ (pathinstate(pfun\ \mathcal{S}[\![s_1]\!]\ \pi_1)\pi_2)\ z$$
$$=$$
$$pfun\ (simplify\ (sequence\mathcal{S}[\![s_1]\!]\ \mathcal{S}[\![s_2]\!]))(\pi_1 \sqcup pathinstate(pfun\ \mathcal{S}[\![s_1]\!]\ \pi_1)\pi_2)z.$$

Also by Lemma 5.3.8(page 153),

$$satisfy\ [\![s_1; s_2]\!]\ [\![p_1; p_2]\!]\ \sigma\ (\pi_1 \sqcup pathinstate(pfun\ \mathcal{S}[\![s_1]\!]\ \pi_1)\pi_2).$$

Therefore

$$\mathcal{M}[\![p_2]\!](\mathcal{M}[\![p_1]\!]\sigma)z$$
$$=$$
$$evalsym\ [\![s_1; s_2]\!]\ [\![p_1; p_2]\!]\ \sigma\ pfun\ (simplify\ (sequence\mathcal{S}[\![s_1]\!]\ \mathcal{S}[\![s_2]\!]))(\pi_1 \sqcup$$
$$pathinstate(pfun\ \mathcal{S}[\![s_1]\!]\ \pi_1)\pi_2)z$$
$$=$$
$$evalsym\ [\![s_1; s_2]\!]\ [\![p_1; p_2]\!]\ \sigma\ pfun\ (\mathcal{S}[\![s_1; s_2]\!])(\pi_1 \sqcup pathinstate(pfun\ \mathcal{S}[\![s_1]\!]\ \pi_1)\pi_2)z$$
by Definition 4.5.5(page 132), as required.

We must now prove the converse. Namely, for all $\pi \in$ *dom* (*pfun* $\mathcal{S}[\![s_1; s_2]\!]$) there exists a program $p_1; p_2 \in [s_1; s_2]$, and a state, σ such that

$$\textit{satisfy}\ [\![s_1; s_2]\!]\ [\![p_1; p_2]\!]\ \sigma\ \pi$$

and

$$\text{for all } z,\ \mathcal{M}[\![p_1; p_2]\!]\sigma\ z = \textit{evalsym}\ [\![s_1; s_2]\!]\ [\![p_1; p_2]\!]\ \sigma\ (\textit{pfun}\ \mathcal{S}[\![s_1; s_2]\!]\ \pi\ z).$$

Proof: Corollary 6.3.1(page 185) states:- for any finite set of predicate symbolic values δ_i obtained from s and any set of values v_i of the right type, there exists a state σ and a program $p \in [s]$ such that *evalsym* $s\ p\ \sigma\ \delta_i = v_i$. The result then follows immediately from the previous part (of which this is the converse).

5.5 Conclusion

The theory in this chapter has led to a proof that the semantics $\mathcal{S}$, of loop–free schemas, introduced in Chapter 4, is both sound and complete.

It is this theorem which provides the essential semantic interpretation that is required in order to justify the algorithms that will be given in Chapter 6 for computing the various dataflow dependencies(Chapter 3) of loop–free schemas.

Chapter 6

Data and Control Dependence in Symbolic Execution Trees

6.1 Introduction

Algorithms for computing *DTLD*, *DTVD*, *DLD* and *DVD* of loop-free schemas are given.

For every loop–free schema s, these algorithms are defined in terms of its symbolic execution tree, $\mathcal{S}[\![s]\!]$.

The fact that $\mathcal{S}[\![s]\!]$ properly characterises $[s]$ enables us to prove that the *DTLD* and *DTVD* algorithms for loop–free schemas are correct provided that the expression syntax of the underlying programming language is sufficiently rich.

The algorithms for computing *DLD* and *DVD* are not proved correct.

In order to compute each of the four dataflow dependencies of a loop–free schema s, two different versions of *data dependence* and four different versions of *control* dependence are defined. These forms of data and control dependence all operate on symbolic execution trees. Each of *DTLD*, *DTVD*, *DLD* and *DVD* is computed by applying the appropriate version of data and control dependence to $\mathcal{S}[\![s]\!]$.

	Data Dependence used	Control Dependence used
DVD	*Vdatadepends*–Definition 6.4.1(page 194)	*Vcontrols*–Definition 6.5.2(page 196)
$DTVD$	*Vdatadepends*	*VTcontrols*–Definition 6.4.3(page 195)
DLD	*Ldatadepends*–Definition 6.2.1(page 173)	*Lcontrols*–Definition 6.5.1(page 196)
$DTLD$	*Ldatadepends*	*LTcontrols*–Definition 6.2.3(page 176)

In Section 6.6, we show that DLD and $DTLD$ can be thought of as special cases of DVD and $DTVD$ respectively. $DTLD$ can be computed by treating the labels as variables, computing the $DTVD$, and then intersecting the final result with the set of all labels.

This means that, in effect, the 'label' and 'variable' versions of each dependence above can be combined into a single dependence. This simplification implies that, in fact, just *one* form of data dependence and two forms of control dependence[1] are all that is required in order to compute the four dataflow dependencies introduced in Chapter 3, when applied to loop–free schemas.

	Data Dependence used	Control Dependence used
DVD	*datadepends*–Definition 6.6.2(page 200)	*controls*–Definition 6.6.4(page 200)
DLD	*datadepends*	*controls*
$DTVD$	*datadepends*	*Tcontrols*–Definition 6.6.3(page 200)
$DTLD$	*datadepends*	*Tcontrols*

We now fully define and prove correct the algorithm for one of the dataflow dependencies, $DTLD$.

6.2 Computing $DTLD$ for Loop–free Schemas

As has just been stated, we require a version of data dependence, which we call *label data dependence* and a version of control dependence which we call *label terminating control dependence*, both operations on symbolic execution trees. These are now defined formally and examples given.

[1] the non–terminating version (*controls*) that we call *Control Dependence* and the terminating version (*Tcontrols*) that we call *Terminating Control Dependence*

6.2.1 Label Data Dependence

Let t be a symbolic execution tree. Intuitively, in each leaf symbolic state, the symbolic value of each variable corresponds to the sequence of assignments that would have to be executed to reach that final state. The set of labels upon which variable, v, is label dependent is precisely the set of labels corresponding to these assignments. Formally,
Variable v *data depends* on label l if and only if there exists σ in the range of $pfun(t)$ such that $l \in labels(\sigma\ v)$. (See Definition 6.2.2(page 173) for the definition of *labels*.)

In the example in Figure 4.6(page 134), c is dataflow label dependent on the set $\{f4, f5\}$ and i on $\{f6\}$. We write $Ldatadepends(t)(x)$ for the set of labels upon which x is label data dependent in t.

Definition 6.2.1 (Label Data Dependence)

$$Ldatadepends(t)(x) = \bigcup_{\sigma \in range(pfun(t))} labels(\sigma\ x)$$

The function, *labels*, which returns the set of labels mentioned in a symbolic value is formally defined for the three types of symbolic value as follows:-

Definition 6.2.2 (*labels*)

$$labels(v) = \emptyset$$

$$labels(f(S)) = \{f\} \cup \bigcup_{d \in S} labels(d)$$

$$labels(\perp) = \emptyset$$

6.2.2 Example of Label Data Dependence

Consider, again, the symbolic execution tree in Figure 6.1(page 174). Its path function is given in Figure 6.2(page 175). The label data dependence is given in Figure 6.3(page 175).

To find the labels upon which variable, v, label data depends, simply collect together the labels of all the symbolic final values of v.

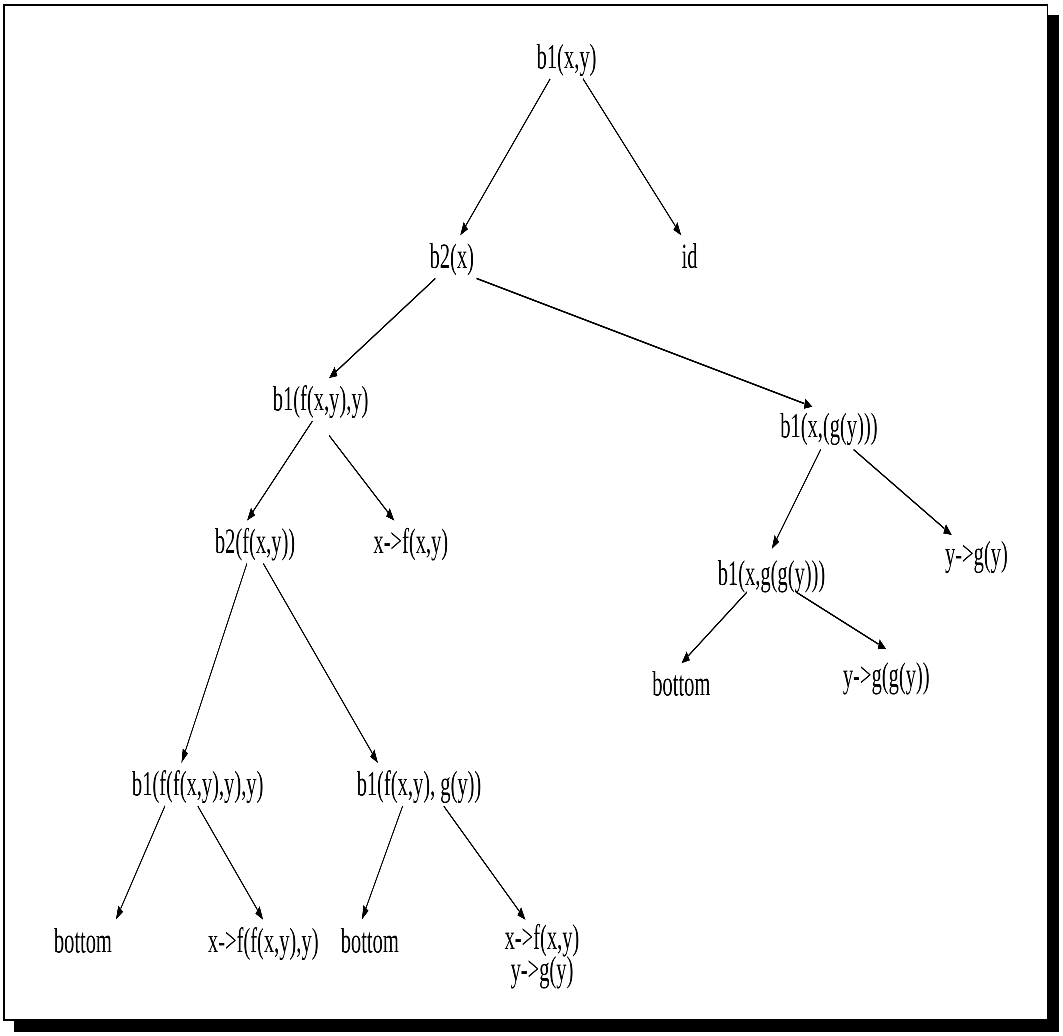


Figure 6.1: Symbolic Execution Tree

If v is label data dependent on l in s, then for some program p in $[s]$, in a given state σ there are an infinite number of possible values for v that can be obtained by replacing the expression at l by another and executing the resulting program in the same state σ.

6.2.3 Label Terminating Control Dependence

Intuitively and informally, v is *label terminating control dependent* on label l in schema s if and only if there is a predicate that depends on l, which 'affects' the final value of v.

True Symbolic Predicates	False Symbolic Predicates	Final State
$\{b1(x,y), b2(x), b1(f(x,y),y), b2(f(x,y)), b1(f(f(x,y),y),y)\}$	$\emptyset$	$\perp$
$\{b1(x,y), b2(x), b1(f(x,y),y), b2(f(x,y))\}$	$\{b1(f(f(x,y),y),y)\}$	$x \mapsto f(f(x,y),y)$
$\{b1(x,y), b2(x), b1(f(x,y),y)), b1(f(x,y),g(y))\}$	$\{b2(f(x,y))\}$	$\perp$
$\{b1(x,y), b2(x), b1(f(x,y),y))\}$	$\{b2(f(x,y)), b1(f(x,y),g(y))\}$	$x \mapsto f(x,y)$ $y \mapsto g(y)$
$\{b1(x,y), b2(x)\}$	$\{b1(f(x,y),g(y))\}$	$x \mapsto f(x,y)$
$\{b1(x,y), b1(x,g(y)), b1(x,g(g(y)))\}$	$\{b2(x)\}$	$\perp$
$\{b1(x,y), b1(x,g(y))\}$	$\{b2(x), b1(x,g(g(y))\}$	$y \mapsto g(g(y))$
$\{b1(x,y)\}$	$\{b2(x), b1(x,g(y))\}$	$y \mapsto g(y)$
$\emptyset$	$\{b1(x,y)\}$	id

Figure 6.2: The Path Function of the symbolic execution tree in Figure 6.1(page 174)

Variable	Labels on which it data depends
x	f
y	g

Figure 6.3: Data Dependence Table

If v is Label Terminating Control Dependent on l in schema s but not Label Data Dependent on l, then given any program p in $[s]$, in a given state σ, there are only a finite number of possible values for v that can be obtained by replacing the expression at l by another and executing the resulting program in the same state, σ. This is because changing the expression at l, does not introduce any new potential final symbolic states, but only potentially affects which one is reached.

Informally, to calculate the set of labels in a symbolic execution tree upon which variable v is label terminating control dependent the following must be done:-
For each pair of non–bottom states, where v has a different final value, work out the set of differences (it cannot be empty, by the 'disagreement lemma': Lemma 5.3.5(page 151)) of the two paths that lead to these two final states i.e. the predicate symbolic values that are true in one path and false in the other. Label l is included in the set of labels upon which v is label terminating control dependent if and only if l occurs as a label in all the symbolic values in this set of differences.

Let t be a symbolic execution tree. Variable x *label terminating control depends* on label l in t if and only if there exist two paths π and π' in the domain of $pfun(t)$ with

$\perp \neq pfun(t)(\pi)x \neq pfun(t)(\pi')x \neq \perp$ such that for all $e \in diffs(\pi, \pi')$, $l \in labels(e)$. (See Definition 5.2.4(page 143) for the definition of *diffs*.)

We write $LTcontrols(t)(v)$ for the set of labels upon which v is label terminating control dependent in t.

Definition 6.2.3 (*LTcontrols*)

$$LTcontrols(t)(v) = \bigcup_{\{(\pi,\pi') \mid \perp \neq pfun(t)(\pi)v \neq pfun(t)(\pi')v \neq \perp\}} \left(\bigcap_{\delta \in diffs(\pi,\pi')} labels\ \delta \right)$$

6.2.4 Examples of Label Terminating Control Dependence

Consider the program $p_{6.4}$, and its corresponding schema $s_{6.4}$, in Figure 6.4(page 177). The symbolic execution tree corresponding to $s_{6.4}$, $\mathcal{S}[\![s_{6.4}]\!]$ has structure as shown in Figure 6.5(page 178). $\mathcal{S}[\![s_{6.4}]\!]$ has four paths. These are shown in Figure 6.6(page 178). There are four pairs of paths in $\mathcal{S}[\![s_{6.4}]\!]$ with different, non-$\perp$, final values for y. These are shown in Figure 6.7(page 178). By Definition 6.2.3(page 176), for each pair of paths (π_i, π_j) with different final values for y, the value of

$$\bigcap_{\delta \in diffs(\pi_i,\pi_j)} labels\ \delta$$

must be calculated. These values are shown in Figure 6.8(page 179). By Definition 6.2.3(page 176), the set of labels upon which variable y is label terminating control dependent is the union of the sets in Figure 6.8(page 179), namely

$$\{b_2\} \cup \emptyset \cup \emptyset \cup \{b_2\}$$

which is $\{b_2\}$. This shows that the predicate $b_2(z)$ in $s_{6.4}$ has an effect on the final value of y in $s_{6.4}$, so b_2 controls y. It also shows that the other predicate in $s_{6.4}$, $b_1(x, z)$, has no effect on the final value for y. This also shows that in $s_{6.4}$, the final value of y is not controlled by the initial value of x but only by the initial value of z.

```
if x<z
then v:=k;
if z<3
then y:=7
else y:=8
```

$if\ b_1(x,z)$
$then\ v := f_2(k);$
$if\ b_2(z)$
$then\ y := f_4()$
$else\ y := f_5()$

Figure 6.4: $p_{6.4}$ and $s_{6.4}$

Now consider the variable v in schema $s_{6.4}$ in Figure 6.4(page 177). There are four pairs of paths in $\mathcal{S}[\![s_{6.4}]\!]$ with different final values for v. These are shown in Figure 6.9(page 179). Again, by Definition 6.2.3(page 176), for each pair of paths (π_i, π_j) with different[2] final values for v, the value of

$$\bigcap_{\delta \in diffs(\pi_i,\pi_j)} labels\ \delta$$

must be calculated. These values are shown in Figure 6.10(page 179). By Definition 6.2.3(page 176), the set of labels upon which variable v is terminating label control dependent is the union of the sets in Figure 6.8(page 179), namely

$$\{b_1, b_2\} \cup \emptyset \cup \emptyset \cup \{b_1, b_2\}$$

which is $\{b_1, b_2\}$. This shows that the predicates $b_1(x,z)$ and $b_2(z)$ in $s_{6.4}$ both have an effect

[2] non–$\perp$

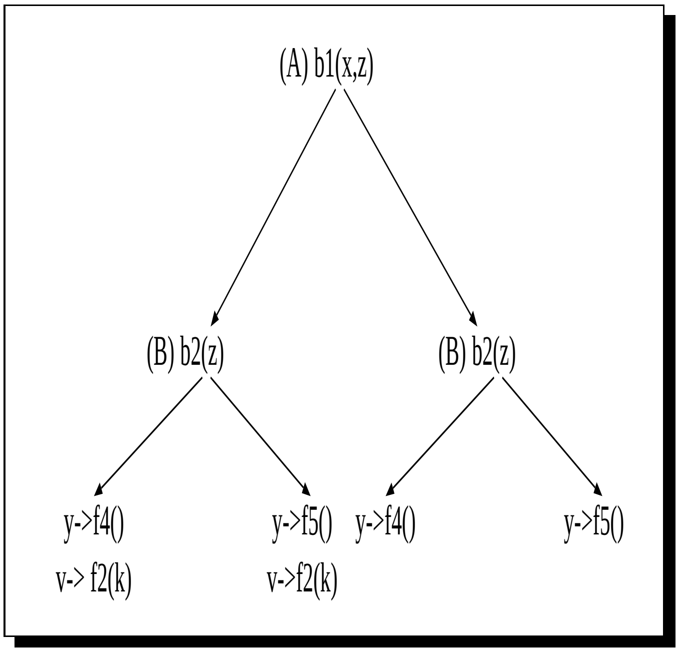


Figure 6.5: The symbolic execution tree, $\mathcal{S}[\![s_{6.4}]\!]$

Path	True Predicates	False Predicates
π_1	$\{A, B\}$	$\{\}$
π_2	$\{A\}$	$\{B\}$
π_3	$\{B\}$	$\{A\}$
π_4	$\{\}$	$\{A, B\}$

Figure 6.6: The four paths of $\mathcal{S}[\![s_{6.4}]\!]$

		Differences
π_1	π_2	$\{B\}$
π_1	π_4	$\{A, B\}$
π_3	π_4	$\{B\}$
π_2	π_3	$\{A, B\}$

Figure 6.7: The Four Pairs of paths of $\mathcal{S}[\![s_{6.4}]\!]$ with different non$-\perp$ final values for y

$\bigcap_{\delta \in diffs(\pi_1,\pi_2)} labels\ \delta$	$=$	$labels(B)$	$=$	$\{b_2\}$
$\bigcap_{\delta \in diffs(\pi_1,\pi_4)} labels\ \delta$	$=$	$labels(A) \cap labels(B)$	$=$	$\emptyset$
$\bigcap_{\delta \in diffs(\pi_3,\pi_4)} labels\ \delta$	$=$	$labels(B)$	$=$	$\{b_2\}$
$\bigcap_{\delta \in diffs(\pi_2,\pi_3)} labels\ \delta$	$=$	$labels(A) \cap labels(B)$	$=$	$\emptyset$

Figure 6.8: The Differences of each pair of paths of $\mathcal{S}[\![s_{6.4}]\!]$ with different non–$\perp$ final values for y

		Differences
π_1	π_3	$\{A\}$
π_1	π_4	$\{A, B\}$
π_2	π_4	$\{A\}$
π_2	π_3	$\{A, B\}$

Figure 6.9: The Four Pairs of paths of $\mathcal{S}[\![s_{6.4}]\!]$ with different non–$\perp$ final values for v

on the final value of v in , so b_1 and b_2 both control v.

This also shows that in $s_{6.4}$, the final value of v is controlled by the initial values of x and z.

An Example with Non–termination

Consider the program $p_{6.11}$, and its corresponding schema $s_{6.11}$, in Figure 6.11(page 180). The symbolic execution tree corresponding to $s_{6.11}$, $\mathcal{S}[\![s_{6.11}]\!]$ has structure as shown in Fig-

$\bigcap_{\delta \in diffs(\pi_1,\pi_3)} labels\ \delta$	$=$	$labels(A)$	$=$	$\{b_1\}$

Figure 6.10: The Differences of each pair of paths of $s_{6.11}$, with different non–$\perp$ final values for v

ure 6.12(page 181). There are *no* paths in $\mathcal{S}[\![s_{6.11}]\!]$ with different non$-\perp$ final values for y. This means there are no labels upon which y is *label terminating control* dependent. There are no terminating programs in $[s_{6.11}]$ where the choice of which 'direction is taken' can make any difference to the final value of y. Put another way, two programs in the same dataflow equivalence class differing only at the expression labelled b_1 cannot both terminate with different[3] final values for y. Similarly for b_2.

For variable v, on the other hand, there are two paths in $\mathcal{S}[\![s_{6.11}]\!]$ with different final values for v. These are π_1 and π_3. (Labelling the paths of $\mathcal{S}[\![s_{6.11}]\!]$ from left to right $\pi_1, \cdots, \pi_4$.) Using Definition 6.2.3(page 176), it can be seen that the set of labels upon which variable v is label terminating control dependent is

$$\bigcap_{\delta \in \mathit{diffs}(\pi_1, \pi_3)} \mathit{labels}\ \delta = \mathit{labels}(A) = \{b_1\}$$

In this case, therefore there are two programs in $[s_{6.11}]$ differing only at label b_1, which in some initial state both terminate with different final values for v.

```
if x<z
then v:=k;
if z<3
then y:=7
else FAIL
```

$$\begin{array}{l} \mathit{if}\ b_1(x,z) \\ \mathit{then}\ v := f_2(k); \\ \mathit{if}\ b_2(z) \\ \mathit{then}\ y := f_4() \\ \mathit{else}\ \mathit{FAIL} \end{array}$$

Figure 6.11: $p_{6.11}$ and $s_{6.11}$

In the example in Figure 4.6(page 134), there are three paths namely, ABD, ABE and AC. Variable c has three different symbolic values at the ends of each of these three paths.

[3] This should remind the reader of the definition of *DTLD*.

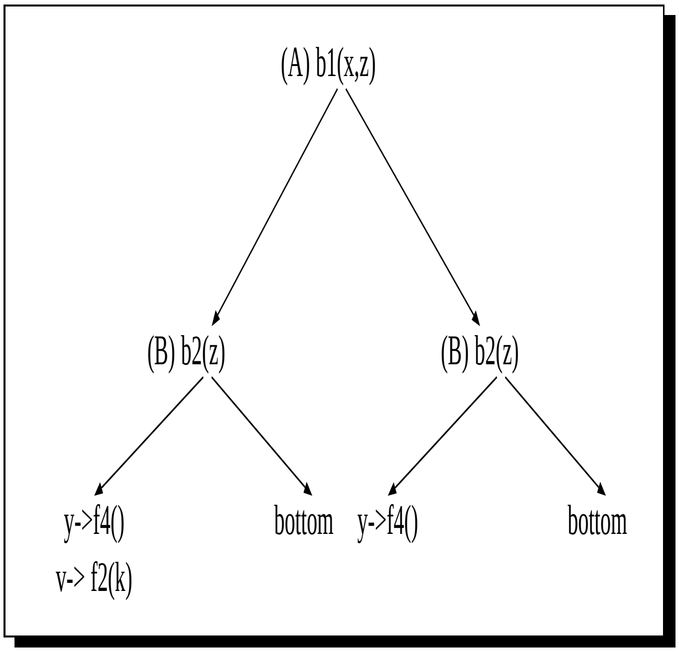


Figure 6.12: The symbolic execution tree, $\mathcal{S}[\![s_{6.11}]\!]$

This gives three 'differences' to calculate

1. differences$(ABD, ABE) = \{B\}$, so variable, c, is label terminating control dependent on $\{f_2, f_3\}$ since $labels(B) = \{f_2, f_3\}$.

2. differences$(ABD, AC) = \{A\}$ so variable, c, is label terminating control dependent on $\{f_1\}$ since $labels(A) = \{f_1\}$.

3. differences$(AC, ABE) = \{A\}$ so variable, c, is label terminating control dependent on $\{f_1\}$ since $labels(A) = \{f_1\}$.

Collecting these together we get c is label terminating control dependent on $\{f_1, f_2, f_3\}$.

Variable i, on the other hand has only two different non–$\perp$ symbolic values at the ends of the three paths (Paths ABD and ABE lead to the same value). In this case, therefore, there are two differences to calculate:

1. differences$(ABD, AC) = \{A\}$, so variable i is label terminating control dependent on $\{f_1\}$ since $labels(A) = \{f_1\}$.

Variable	Label Data Dependence	Label Terminating Control Dependence
c	$\{f_4, f_5\}$	$\{f_1, f_2, f_3\}$
i	$\{f_6\}$	$\{f_1\}$

Figure 6.13: Label Data Dependence and Label Terminating Label Dependence

c	$\{f_1, f_2, f_3, f_4, f_5\}$
i	$\{f_1, f_6\}$

Figure 6.14: DTLslice

2. differences$(AC, ABE) = \{A\}$, so variable, i, is label terminating control dependent on $\{f_1\}$ since $labels(A) = \{f_1\}$.

This shows that i is label terminating control dependent on $\{f_1\}$.

In the example in Figure 6.1(page 174), in order to compute the set of labels upon which the variable x is label terminating control dependent, it can be seen that symbolic predicate values that can cause different terminating final values of x are:- $b_1(x, y), b_2(x)$, $b_1(f(x, y), y)$ and $b_2(f(x, y))$. Variable x is thus label terminating control dependent on the set of labels $\{b_1, b_2, f\}$.

6.2.5 The DTLslice of a Symbolic Execution Tree

Definition 6.2.4 (DTLslice)

Let t be a Symbolic Execution Tree. The DTLslice for variable v in t is defined to be the union of label data dependence and label terminating control dependence of v, i.e.

$$\text{DTLslice}(t)(v) = \mathit{Ldatadepends}(t)(v) \cup \mathit{LTcontrols}(t)(v)$$

The analysis of $s_{4.5}$ is given in Figure 6.13(page 182) and Figure 6.14(page 182).

6.2.6 The Algorithm for *DTLD*

In order to compute the set of labels l for which x *DTLD* l in loop–free schema s, s is first translated into its symbolic execution tree, $\mathcal{S}[\![s]\!]$ and then the DTLslice (Definition 6.2.4(page 182)) of x with respect to $\mathcal{S}[\![s]\!]$ is computed.

6.3 Correctness of the Algorithm for *DTLD*

In this section, some results and definitions are provided. These are needed in order to prove the main correctness theorem of this chapter.

Definition 6.3.1 (*varof*)
Since labels are unique, if the outermost label of symbolic value, f_i occurs as the label of an expression in an assignment to x, say, then by uniqueness, it cannot occur as the outermost label of an assignment to any other variable. The value of all symbolic values whose outermost label is f_i must be therefore associated with the variable x and only the variable x. Given a schema s and symbolic value $f_i(S)$ we define *varof* s $f_i(S)$ to be the variable associated with f_i as just defined.

For a symbolic value that is the variable, v, *varof* s $v = v$.

Definition 6.3.2 (obtained from)

1. For all variables v and schemas s, the symbolic value that is the variable v is *obtained from* schema s.

2. The symbolic value $f(S)$ is *obtained from* the schema s if and only if

   (a) all δ in S are obtained from s and

   (b) there is a symbolic expression $f(T)$ in s, such that for *varof* s is a one to one correspondence between S and T, i.e. for each variable v in T, there is a unique symbolic value δ in S such that *varof* $\delta = v$. Conversely, for each δ in S there is a unique variable v in T such that *varof* $\delta = v$.

Definition 6.3.3 (Assigned Symbolic Value)
Let s be a schema and δ=$f(S)$ be a symbolic value obtained from s. δ is an assigned symbolic value if and only if f is the label of an expression occurring on the right hand side of an assignment in s[4].

[4] f cannot, therefore, also be the label of a predicate, since labels are unique.

Definition 6.3.4 (Predicate Symbolic Value)

Let s be a schema and $\delta = f(S)$ be a symbolic value obtained from s. δ is a predicate symbolic value if and only if f is the label of a predicate in s.

Theorem 6.3.1 (Outermost Labels of Predicates Symbolic Values) *Given a schema s, for all predicate symbolic values, δ, obtained from s, the outermost label of δ does not occur as non–outermost label of any symbolic value obtained from s.*

Proof: obvious.

Theorem 6.3.2 (Outermost Labels of Predicate Symbolic Values) *Given a loop–free schema s, for all predicate symbolic values, δ, obtained from s, for all variables, x, the outermost label of δ does not occur in pfun $\mathcal{S}[\![s]\!]\ x$.*

Proof: obvious.

Theorem 6.3.3 *Let δ_i, $i \in \{1 \cdots n\}$ be a set of distinct symbolic values obtained from s that are either variables or assigned symbolic values and let v_i $i \in \{1 \cdots n\}$ be a set of n distinct integers. Then there exists a program p in $[s]$ and a state σ such that for all $i \in \{1 \cdots n\}$, $evalsym\ s\ p\ \sigma\ \delta_i = v_i$.*

Proof:

Induction on the maximum depth of the δ_i.

Base Case

The δ_i are all variables. Simply pick σ so that $\sigma\ \delta_i = v_i$

Induction Hypothesis Assume true for all δ_i of depth $< m$. Let δ_i, $i \in \{1 \cdots n\}$ have maximum depth of m. Then since all the δ_i, are unique, and

$$evalsym\ s\ p\ \sigma\ f_i(S) = \mathcal{E}[\![p\ f_i]\!] \bigcup_{\delta \in S} varof(s, \delta) \mapsto (evalsym\ s\ p\ \sigma\ \delta)$$

If f_i, occurs more than once as the outermost label of any other δ_i then by induction hypothesis the states:

$$\bigcup_{\delta \in S} varof(s, \delta) \mapsto (evalsym\ s\ p\ \sigma\ \delta)$$

will be unique. By Assumption 3.4.1, the expressions in p corresponding to f_i can be chosen as required.

Corollary 6.3.1 *Let δ_i, $i \in \{1 \cdots n\}$ be a set of distinct predicate symbolic values obtained from s and let b_i $i \in \{1 \cdots n\}$ be a set of boolean values. Then there exists a program p in $[s]$ and a state σ such that for all $i \in \{1 \cdots n\}$, evalsym s p σ $\delta_i = b_i$.*

Proof: follows immediately from Assumption 3.4.1, Theorem 6.3.3(page 184) and Theorem 6.3.2(page 184).

Theorem 6.3.4 *Given a loop–free schema s and a finite set S, of symbolic values (that are not predicate symbolic values) obtained from s. Given a label, f, there exists a state σ and programs p and p' differing only at f such that for all sub–symbolic values δ_i, δ_j, of all elements of S*

$$\delta_i \neq \delta_j \implies \mathit{evalsym}\ s\ p\ \sigma\ \delta_i \neq \mathit{evalsym}\ s\ p\ \sigma\ \delta_j$$
$$\mathit{and}$$
$$\delta_i \neq \delta_j \implies \mathit{evalsym}\ s\ p'\ \sigma\ \delta_i \neq \mathit{evalsym}\ s\ p'\ \sigma\ \delta_j$$
$$\mathit{and}$$
$$f \in \mathit{labels}\ \delta_i \implies \mathit{evalsym}\ s\ p\ \sigma\ \delta_i \neq \mathit{evalsym}\ s\ p'\ \sigma\ \delta_i$$
$$\mathit{and}$$
$$f \notin \mathit{labels}\ \delta_i \implies \mathit{evalsym}\ s\ p\ \sigma\ \delta_i = \mathit{evalsym}\ s\ p'\ \sigma\ \delta_i$$

Proof:
Induction on the maximum depth of each element of S
<u>BaseCase</u>
If the maximum depth of S is zero then every element of S is a variable. Trivial.

Now assume the maximum depth of S is $N > 0$.
Consider the set T of all sub–symbolic values of all the elements of S whose depth is less than N. By the induction hypothesis, there exists a state σ and programs p and p' differing only at f such that for all sub–symbolic values δ_i, δ_j of all elements, δ of T

$$\delta_i \neq \delta_j \implies \mathit{evalsym}\ s\ p\ \sigma\ \delta_i \neq \mathit{evalsym}\ s\ p\ \sigma\ \delta_j$$
$$\text{and}$$

$$\delta_i \neq \delta_j \implies \mathit{evalsym}\ s\ p'\ \sigma\ \delta_i \neq \mathit{evalsym}\ s\ p'\ \sigma\ \delta_j$$

and

$$f \in \mathit{labels}\ \delta_i \implies \mathit{evalsym}\ s\ p\ \sigma\ \delta_i \neq \mathit{evalsym}\ s\ p'\ \sigma\ \delta_i$$

and

$$f \notin \mathit{labels}\ \delta_i \implies \mathit{evalsym}\ s\ p\ \sigma\ \delta_i = \mathit{evalsym}\ s\ p'\ \sigma\ \delta_i$$

Consider all the elements $f_i(S_i)$ of S of depth N. We work through them one at a time. Take the 'first' one, $f_1(S_1)$. Then

$$\bigcup_{\delta \in S_1} (\mathit{varof}\ s\ \delta) \mapsto (\mathit{evalsym}\ s\ p\ \sigma\ \delta)$$

is a state that has not occurred in the evaluation in p of any other symbolic value whose outermost label is f_1. Similarly

$$\bigcup_{\delta \in S_1} (\mathit{varof}\ s\ \delta) \mapsto (\mathit{evalsym}\ s\ p'\ \sigma\ \delta)$$

is a state that has not occurred in the evaluation in p' of any other symbolic value whose outermost label is f_1.

If $f_1 \neq f$, the we want the same expression $p(f_1)$ (using the notation discussed in Section 3.11.3) in its place in both p and p'.
If $f \notin \mathit{labels}\ f_1(S_1)$ we require

$$\mathit{evalsym}\ s\ p\ \sigma\ f_1(S_1) = \mathit{evalsym}\ s\ p'\ \sigma\ f_1(S_1).$$

By Assumption 3.4.1, we can replace e_1 in p and p', if necessary by a new expression, e'_1 such that they agree on all previous states and also

$$\mathit{evalsym}\ s\ p\ \sigma\ f_1(S_1) = \mathit{evalsym}\ s\ p'\ \sigma\ f_1(S_1)$$

and so that

$$\mathit{evalsym}\ s\ p\ \sigma\ f_i(S_i)$$

is different from any other value so far encountered in evaluating symbolic values with respect to p and

$$evalsym\ s\ p'\ \sigma\ f_i(S_i)$$

is different from any other value so far encountered in evaluating symbolic values with respect to p'.

If, on the other hand, $f \in labels\ f_1\,(S_1)$, we require

$$evalsym\ s\ p\ \sigma\ f_1(S_1) \neq evalsym\ s\ p'\ \sigma\ f_1(S_1).$$

By Assumption 3.4.1, we can replace e_1 in p and p', if necessary by an new expression, e'_1 such that they agree on all previous states and also

$$evalsym\ s\ p\ \sigma\ f_1(S_1) \neq evalsym\ s\ p'\ \sigma\ f_1(S_1)$$

and so that

$$evalsym\ s\ p\ \sigma\ f_i(S_i)$$

is different from any other value so far encountered in evaluating symbolic values with respect to p and

$$evalsym\ s\ p'\ \sigma\ f_i(S_i)$$

is different from any other value so far encountered in evaluating symbolic values with respect to p'.

The final possibility is that $f_1 = f$.

In this case we require

$$evalsym\ s\ p\ \sigma\ f_1(S_1) \neq evalsym\ s\ p'\ \sigma\ f_1(S_1).$$

Again, since the states in which $p\ f$ and $p\ f'$ have not been previously encountered, we can again choose new values for $p\ f$ and $p\ f'$ such that they are the same on all previous states and differ on the new ones.

Repeat this process until all symbolic values of depth N have been processed. We will then be left with two programs q and q', say, such that for all sub–symbolic values δ_i, δ_j of all elements of S

$$\delta_i \neq \delta_j \implies evalsym\ s\ q\ \sigma\ \delta_i \neq evalsym\ s\ q\ \sigma\ \delta_j$$
$$\text{and}$$
$$\delta_i \neq \delta_j \implies evalsym\ s\ q'\ \sigma\ \delta_i \neq evalsym\ s\ q'\ \sigma\ \delta_j$$
$$\text{and}$$
$$f \in labels\ \delta_i \implies evalsym\ s\ q\ \sigma\ \delta_i \neq evalsym\ s\ q'\ \sigma\ \delta_i$$
$$\text{and}$$
$$f \notin labels\ \delta_i \implies evalsym\ s\ q\ \sigma\ \delta_i = evalsym\ s\ q'\ \sigma\ \delta_i$$

as required.

This completes the proof of Theorem 6.3.4(page 185).

6.3.1 Proof of *DTLD* Algorithm

We are now in a position to prove the main theorem which states that given a loop free schema s, the set of labels upon which variable x is dataflow terminating label dependent can be computed by translating s into a symbolic execution tree, t using the semantic function $\mathcal{S}$ defined in Chapter 4 and then computing the DTLslice of t using label data dependency and label terminating control dependency described in Section 6.1 of this chapter.

Theorem 6.3.5 *Given a loop-free schema s,*

$$l \in DTLD\ s\ x \iff l\ \in\ \text{DTLslice}\ \mathcal{S}[\![s]\!]\ x$$

Proof:

We must show that there exist two programs p and p' in $[s]$ differing only at label l in s and a state σ such that

$$\perp \neq \mathcal{M}[\![p]\!]\sigma\ x \neq \mathcal{M}[\![p']\!]\sigma\ x \neq \perp$$
$$\Longleftrightarrow$$
$$l\ \in\ \text{DTLslice}\ \mathcal{S}[\![s]\!]\ x.$$

$\Longrightarrow$

Assume that there exist two programs p_1 and p_2 in $[s]$ differing only at label l in s and a state σ such that $\perp \neq \mathcal{M}[\![p]\!]\sigma\ x \neq \mathcal{M}[\![p']\!]\sigma\ x \neq \perp$. By Theorem 5.4.1(page 160), there exist unique paths π and π' in *dom* (*pfun* s) such that

$$evalsym\ s\ p\ \sigma\ (pfun\ s\ \pi\ x) \neq evalsym\ s\ p'\ \sigma\ (pfun\ s\ \pi'\ x)$$

such that

$$satisfy\ s\ p\ \sigma\ \pi \text{ and } satisfy\ s\ p'\sigma\ \pi'.$$

<u>Case 1</u> if $\pi = \pi'$ then $y \in labels(pfun\ s\ \pi\ x)$ since p_1 and p_2 in $[s]$ differ only at label l. (Otherwise $evalsym\ s\ p\ \sigma\ (pfun\ s\ \pi\ x)$ and $evalsym\ s\ p'\ \sigma\ (pfun\ s\ \pi\ x)$ would have to be identical).

therefore $l \in Ldatadepends\ s\ x$ as required.

<u>Case 2</u> if $\pi \neq \pi'$

then, again, if $(pfun\ s\ \pi\ x) = (pfun\ s\ \pi'\ x)$ then either $l \in labels(pfun\ s\ \pi\ x)$ or $l \in labels(pfun\ s\ \pi'\ x)$ so $l \in$ DTLslice $s\ x$, as before.

Assume $(pfun\ s\ \pi\ x) \neq (pfun\ s\ \pi'\ x)$ and $l \notin labels((pfun\ s\ \pi\ x))$ and $l \notin labels((pfun\ s\ \pi'\ x))$. For all $\delta \in diffs(\pi, \pi')$, $l \in labels\ \delta$ and therefore l is in *LTcontrols* $s\ x$ and hence in DTLslice $s\ x$, as required.

$\Longleftarrow$

Assume $l \in$ DTLslice $s\ x$.

Case 1 If $l \in \mathit{Ldatadepends}\ s\ x$

then there must exist a path $\pi = (\pi_t, \pi_f)$ with $(\mathit{pfun}\ s\ \pi\ x)$ such that $l \in \mathit{labels}(\mathit{pfun}\ s\ \pi\ x)$. By Theorem 6.3.4(page 185), we can pick a state σ and programs p and p' differing only at l such that for all sub–symbolic values δ_i ,δ_j in $\pi_t \cup \pi_f \cup \{(\mathit{pfun}\ s\ \pi\ x)\}$ such that

$$\delta_i \neq \delta_j \implies \mathit{evalsym}\ s\ p\ \sigma\ \delta_i \neq \mathit{evalsym}\ s\ p'\ \sigma\ \delta_j$$
$$\text{and}$$
$$\delta_i \neq \delta_j \implies \mathit{evalsym}\ s\ p'\ \sigma\ \delta_i \neq \mathit{evalsym}\ s\ p'\ \sigma\ \delta_j$$
$$\text{and}$$
$$l \in \mathit{labels}\ \delta_i \implies \mathit{evalsym}\ s\ p\ \sigma\ \delta_i \neq \mathit{evalsym}\ s\ p'\ \sigma\ \delta_i.$$

Since this means that state will be different when evaluating symbolic predicates with the same outermost label, by Assumption 3.4.1, we can find values of each predicate expression such that

$$(\mathit{satisfy}\ s\ p\ \sigma\ \pi)$$
$$\text{and } (\mathit{satisfy}\ s\ p'\ \sigma\ \pi')$$

but

$$\mathit{evalsym}\ s\ p\ \sigma\ (\mathit{pfun}\ s\ \pi\ x) \neq \mathit{evalsym}\ s\ p'\ \sigma(\mathit{pfun}\ s\ \pi\ x).$$

since $l \in \mathit{labels}(\mathit{pfun}\ s\ \pi\ x)$, so by Theorem 5.4.1(page 160),

$$\mathcal{M}[\![p]\!]\sigma\ x \neq \mathcal{M}[\![p']\!]\sigma\ x.$$

Also, since $l \in \mathit{labels}(\mathit{pfun}\ s\ \pi\ x)$, $s(\mathit{pfun}\ s\ \pi\ x) \neq \perp$
so $\perp \neq \mathit{evalsym}\ s\ p\ \sigma\ (\mathit{pfun}\ s\ \pi\ x) \neq \mathit{evalsym}\ s\ p'\ \sigma(\mathit{pfun}\ s\ \pi\ x) \neq \perp$.
Therefore

$$\perp \neq \mathcal{M}[\![p]\!]\sigma\ x \neq \mathcal{M}[\![p']\!]\sigma\ x \neq \perp.$$

and p and p' have been chosen to differ only at l, as required.

Case 2 If $l \in \mathit{LTcontrols}\ s\ x$

then there exist two paths π and π' with $l \in labels\delta$ for all $\delta \in diffs(\pi, \pi') \neq \emptyset$ such that

$$pfun\ s\ \pi\ x \neq pfun\ s\ \pi'\ x.$$

We can assume that neither $l \notin labels(pfun\ s\ \pi\ x)$ and $l \notin labels(pfun\ s\ \pi'\ x)$ since otherwise $l \in (Ldatadepends\ s\ x)$ which we have already considered.

<u>Case 1</u> if l is a predicate label.
So l must be the outermost label of each element of $diffs(\pi, \pi')$.
So each element of $diffs(\pi, \pi')$ must be of the form $l(S_i)$ and none of the S_i mention l.
By Theorem 6.3.4(page 185), there exists a state σ and a program p such that for all δ_i, δ_j in

$$\{(pfun\ s\ \pi\ x)\} \cup \{(pfun\ s\ \pi'\ x)\} \cup \bigcup_{f_i(S_i) \in \pi \cup \pi'} S_i,$$

$$\delta_i \neq \delta_j \implies evalsym\ s\ p\ \sigma\ \delta_i \neq evalsym\ s\ p\ \sigma\ \delta_j.$$

So

$$i \neq j \implies \bigcup_{\delta \in S_i} (varof\ \ s\ \delta) \mapsto (evalsym\ s\ p\ \sigma\ \delta) \neq \bigcup_{\delta \in S_j} (varof\ \ s\ \delta) \mapsto (evalsym\ s\ p\ \sigma\ \delta)$$

As in the previous proof, we can work our way through the elements of $\pi \cup \pi'$, choosing the value of expression corresponding to the outermost label so that it gives us the required values (Assumption 3.4.1), and does not disagree with all previously encountered states, to give us two programs p and p' so that

$$satisfy\ s\ p\ \sigma\ \pi$$

and

$$satisfy\ s\ p'\ \sigma\ \pi'.$$

By Theorem 5.4.1(page 160),

$$\mathcal{M}[\![p]\!]\sigma\ x = evalsym\ s\ p\ \sigma\ (pfun\ s\ \pi\ x) \neq \perp$$

and

$$\mathcal{M}[\![p']\!]\sigma\ x = evalsym\ s\ p'\ \sigma\ (pfun\ s\ \pi'\ x) \neq \perp.$$

But

$$evalsym\ s\ p\ \sigma\ (pfun\ s\ \pi\ x) \neq evalsym\ s\ p'\ \sigma\ (pfun\ s\ \pi'\ x)$$

and so

$$\perp \neq \mathcal{M}[\![p]\!]\sigma\ x \neq \mathcal{M}[\![p']\!]\sigma\ x \neq \perp$$

and p and p' differ only at l as required.

<u>Case 2</u> l is a not a predicate label,

then, by Theorem 6.3.4(page 185), there exists a state σ and a programs p and p' differing only at l such that for all δ_i, δ_j in

$$\{(pfun\ s\ \pi\ x)\} \cup \{(pfun\ s\ \pi'\ x)\} \cup \bigcup_{f_i(S_i)\in\pi\cup\pi'} S_i,$$

$$\begin{gathered}
\delta_i \neq \delta_j \implies evalsym\ s\ p\ \sigma\ \delta_i \neq evalsym\ s\ p\ \sigma\ \delta_j \\
\text{and} \\
\delta_i \neq \delta_j \implies evalsym\ s\ p'\ \sigma\ \delta_i \neq evalsym\ s\ p'\ \sigma\ \delta_j \\
\text{and} \\
l \in labels\ \delta_i \implies evalsym\ s\ p\ \sigma\ \delta_i \neq evalsym\ s\ p'\ \sigma\ \delta_i \\
\text{and} \\
l \notin labels\ \delta_i \implies evalsym\ s\ p\ \sigma\ \delta_i = evalsym\ s\ p'\ \sigma\ \delta_i.
\end{gathered}$$

Again, as in the previous proof, we can therefore work our way through the elements of $\pi \cup \pi'$ choosing the value of expression corresponding to the outermost label so it gives us the required values (by Assumption 3.4.1), and does not disagree with all previously encountered states, to give us two programs p and p' so that

$$\begin{gathered}
satisfy\ s\ p\ \sigma\ \pi \\
\text{and} \\
satisfy\ s\ p'\ \sigma\ \pi'.
\end{gathered}$$

By Theorem 5.4.1(page 160),

$$\mathcal{M}[\![p]\!]\sigma\ x = evalsym\ s\ p\ \sigma\ (pfun\ s\ \pi\ x)$$

and

$$\mathcal{M}[\![p']\!]\sigma\ x = evalsym\ s\ p'\ \sigma\ (pfun\ s\ \pi'\ x).$$

But

$$\perp \neq evalsym\ s\ p\ \sigma\ (pfun\ s\ \pi\ x) \neq evalsym\ s\ p'\ \sigma\ (pfun\ s\ \pi'\ x) \neq \perp$$

So

$$\perp \neq \mathcal{M}[\![p]\!]\sigma\ x \neq \mathcal{M}[\![p']\!]\sigma\ x \neq \perp$$

and p and p' differ only at l as required.
Theorem 6.3.4 shows that for all loop free schemas s, the *DTLD* of s can be calculated by computing the DTLslice of $\mathcal{S}[\![s]\!]$.

This completes the proof of correctness of the *DTLD* algorithm. We now give a similar algorithm for *DTVD*. Due to its similarity to the previous example, it is not covered in so much detail and the proof is relegated to the appendix.

6.4 Computing *DTVD* for Loop–free Schemas

The dataflow (terminating) variable dependence of symbolic execution tree, t, is similarly computed using $pfun(t)$(Definition 4.4.2(page 125)). It is the union of two *smaller* dependencies: *variable data dependence* and *variable terminating control dependence.*

6.4.1 Variable Data Dependence

Let t be a symbolic execution tree. Informally, v *variable data depends* on variable x in t if there is a leaf state of t where variable v gets mapped to a symbolic value that mentions the variable x. This means that there is a path through the program where the final value of v is computed using a sequence of assignments where the final assignment to v is an expression which depends upon the initial value of x. This corresponds exactly to traditional data dependence.

Formally, variable v *variable data depends* on variable x if and only if there exists σ in the range of $pfun(t)$ such that $x \in variables(\sigma\ v)$. (See Definition 6.4.2(page 194) for the definition of *variables*)

We write $Vdatadepends(t)(x)$ for the set of variables upon which x is variable data dependent in t.

Definition 6.4.1 (Variable Data Dependence)

$$Vdatadepends(t)(x) = \bigcup_{\sigma \in range(pfun(t))} variables(\sigma\ x)$$

The function,*variables*, which returns the set of variables mentioned in a symbolic value is formally defined for the three types of symbolic value as follows:-

Definition 6.4.2 (*variables*)

$$variables(v) = \{v\}$$

$$variables(f(S)) = \bigcup_{d \in S} variables(d)$$

$$variables(\perp) = \emptyset$$

6.4.2 Variable Terminating Control Dependence

Let t be a symbolic execution tree. Variable x *variable control depends* on variable v in t if and only if there exist two paths π and π' in the domain of $pfun(t)$ with $\perp \neq pfun(t)(\pi)x \neq pfun(t)(\pi')x \neq \perp$ such that for all $e \in diffs(\pi, \pi')$, $v \in variables(e)$. (See Definition 5.2.4(page 143) for the definition of *diffs*.) Informally this means that we can find two states differing only on variable v such in one state one path is chosen and in the other state the other path is chosen. Since the symbolic states at the end of these paths have different values for x we can choose two initial states differing only on variable v where the symbolic values correspond to different 'real' values for x.

We write $VTcontrols(t)(v)$ for the set of variables upon which v is control dependent in t.

Definition 6.4.3 (*VTcontrols*)

$$VTcontrols(t)(v) = \bigcup_{\{(\pi,\pi')\mid \perp\neq pfun(t)(\pi)\,v\neq pfun(t)(\pi')\,v\neq\perp\}} \left(\bigcap_{\delta\in diffs(\pi,\pi')} variables\ \delta\right)$$

6.4.3 The DTVslice of a Symbolic Execution Tree

Definition 6.4.4 (DTVslice)

Let t be a Symbolic Execution Tree. The DTVslice for variable v in t is the union of variable data dependence and variable terminating control dependence of v. i.e.

$$\text{DTVslice}(t)(v) = Vdatadepends(t)(v) \cup VTcontrols(t)(v)$$

6.4.4 The Algorithm for *DTVD*

In order to compute the set of variables v for which x *DTLD* v in loop–free schema s, s is first translated into its symbolic execution tree, $\mathcal{S}[\![s]\!]$ and then the DTVslice (Definition 6.4.4(page 195)) of x with respect to $\mathcal{S}[\![s]\!]$ is computed.

The proof of this algorithm is given in Appendix D page 281.

As can be seen the algorithm for computing *DTVD* for Loop–free Schemas and its proof are almost identical to that for *DTLD*. One uses variables and the other uses labels. In view of the discussion that follows (Section 6.6) this is not surprising. We show that to compute *DTLD* we can think of labels as variables, compute the *DTVD*, and intersect the result with the set of labels. Labels are just a special kind of variable.

6.5 The Algorithms for *DLD* and *DVD*

We claim, but do not prove, that in order to produce algorithms for the non–terminating dependency relation *DVD* and hence *DLD*, all that is required is a small change in the definitions of Control Dependence. These results are left to 'future work'.

		Differences
π_1	π_2	$\{B\}$
π_1	π_4	$\{A, B\}$
π_2	π_3	$\{A, B\}$
π_3	π_4	$\{B\}$

Figure 6.15: The Four Pairs of paths of $\mathcal{S}[\![s_{6.11}]\!]$ with different final values for y

Definition 6.5.1 (*Lcontrols*)

$$Lcontrols(t)(v) = \bigcup_{\{(\pi,\pi')\,|\,pfun(t)(\pi)\,v \neq pfun(t)(\pi')\,v\}} \left(\bigcap_{\delta \in diffs(\pi,\pi')} labels\ \delta \right)$$

Definition 6.5.2 (*Vcontrols*)

$$Vcontrols(t)(v) = \bigcup_{\{(\pi,\pi')\,|\,pfun(t)(\pi)\,v \neq pfun(t)(\pi')\,v\}} \left(\bigcap_{\delta \in diffs(\pi,\pi')} variables\ \delta \right)$$

The only difference between terminating and non–terminating control dependency is that, in the latter, two paths are considered to lead to different values of v if using one path we get bottom (i.e non-termination) and in terminating dependence, neither path was allowed to lead to bottom. This exactly captures the difference between the terminating and non–terminating dependence defined in Chapter 3.

6.5.1 Example

Consider, again, the schema $s_{6.11}$ in Figure 6.11(page 180). Although there we *no* pairs of paths with different non–$\perp$ values for y the same is not true when the condition that the values must not be $\perp$ is dropped. In this case, there are four pairs of paths with different values for y. These are given in Figure 6.15(page 196).

$\bigcap_{\delta \in diffs(\pi_1,\pi_2)} labels\ \delta$	$=$	$labels(A)$	$=$	$\{b_2\}$
$\bigcap_{\delta \in diffs(\pi_1,\pi_4)} labels\ \delta$	$=$	$labels(A) \cap labels(B)$	$=$	$\emptyset$
$\bigcap_{\delta \in diffs(\pi_2,\pi_3)} labels\ \delta$	$=$	$labels(A) \cap labels(B)$	$=$	$\emptyset$
$\bigcap_{\delta \in diffs(\pi_3,\pi_4)} labels\ \delta$	$=$	$labels(B)$	$=$	$\{b_2\}$

Figure 6.16: The Four Pairs of paths of $\mathcal{S}[\![s_{6.11}]\!]$ with different final values for y

By Definition 6.5.1(page 196), the set of labels upon which variable y is label control dependent is the union of the sets in Figure 6.16(page 197), namely

$$\{b_2\} \cup \emptyset \cup \emptyset \cup \{b_2\}$$

which is $\{b_2\}$. This shows that the predicate $b_2(z)$ in $s_{6.4}$ has an effect (using this definition) on the final value of y in $s_{6.11}$, since in this form of label dependence (DLD), non–termination is considered a different value from the non–terminating value. We have shown that although b_2 does not 'label terminating control' y in $s_{6.11}$, it does 'label control' y in $s_{6.11}$.

6.6 Labels are really Variables

In this section, we show how dataflow label dependence can be translated into dataflow variable dependence. We claim that given a schema s if to each expression we simply add a new unique label from some set, L, of labels, and calculate the set of variables upon which each variable is dataflow variable dependent and then intersect this set with L we will get the dataflow label dependence of each variable. In other words, we can think of the outer label of each labelled expression in a schema as just another variable.

6.6.1 Example

For example consider Figure 6.17(page 198). Here we have augmented schema $s_{3.22}$ of Figure 3.22(page 109). We then calculate the variable dependence of this new augmented schema to get the results shown in Figure 6.18(page 198). The range of this relation is then restricted to just labels to give us the results shown in Figure 6.19(page 198).

while $f_1(f_1, i)$
do
 begin
  if $f_2(f_2, c)$
  then
  begin
   $c := f_3(f_3, y);$
   $x := f_4(f_4)$
  end;
  $i := f_5(f_5, i)$
 end

Figure 6.17: Adding Extra Variables for Label Dependence

Variable	$DTVD$
x	$f_1\ i\ f_2\ y\ f_4$
c	$f_1\ i\ f_2\ f_3$
i	$f_1\ i\ f_5$

Figure 6.18: Adding Extra Variables for Label Dependence

Variable	$DTLD$
x	$f_1\ f_2\ f_4$
c	$f_1\ f_2\ f_3$
i	$f_1\ f_5$

Figure 6.19: Label Dependence

Each added variable does not occur on the left hand side of any assignment and occurs only once in each schema. The reason this works is that for any finite set of variables V, there are countably many expressions that reference V. We can think of this extra variable as the 'index' into this countable set of expressions.

6.6.2 Justification of Label Adding

We prove that these extra label variables do not *interfere* with Dataflow Variable Dependence of the original schema:

Given a program p, let the original schema be s and the label added schema be s', then computing the Dataflow Variable Dependence of s is the same as computing Dataflow Variable Dependence of s' and then ignoring the labels. More formally,

Theorem 6.6.1 *For any program, p, let the corresponding 'non–label added' and 'label added' schemas be called s and s' respectively. Suppose the set of all added labels is L. We claim that for all variables x, x DVD y in $s \iff x$ DVD y in s' and $y \notin L$.*

Proof: Suppose s contains the expressions $e_1, \cdots, e_n$ labelled $l_1, \cdots, l_n$. So s' contains the expressions $e_1 \cup \{l_1\}, \cdots, e_n \cup \{l_n\}$. We use the following properties about expressions:

Property 6.6.1

For every expression, e such that e *VD* S, for all values k, there exists an expression e' such that e' *VD* $S \cup \{x\}$ and for all states σ with $x = k$, $\mathcal{E}[\![e]\!]\sigma = \mathcal{E}[\![e']\!]\sigma$. For example, if $e = y + 1$ and $k = 79$, then put $e' = y + 80 - x$. Then in all states with $x = 79$, e and e' are equal.

Property 6.6.2

For every expression, e, such that e *VD* S, for all values k, there exists an expression e' such that e' *VD* $S - \{x\}$ and for all states σ with $x = k$, $\mathcal{E}[\![e]\!]\sigma = \mathcal{E}[\![e']\!]\sigma$. For example, if $e = x + 1$ and $k = 79$, then put $e' = 80$. Then in all states where $x = 79$, e and e' are identical.

Consider any program, q in $[s]$. Suppose q contains (proper) expressions $d_1, \cdots, d_n$. Since all expressions are uniquely labelled in s', by property 6.6.1, there exists a program, q' containing (proper) expressions $d'_1, \cdots, d'_n$ in $[s']$ and values $v_1, \cdots, v_n$ for $l_1, \cdots, l_n$ such that for all states σ, with $l_i = v_i$ for all i, $\mathcal{E}[\![d_i]\!]\sigma = \mathcal{E}[\![d'_i]\!]\sigma$. Since q' cannot change the values of the l_i (the l_i have been chosen in that way), then in all the states, σ with $l_i = v_i$ we must have $\mathcal{M}[\![q]\!]\sigma = \mathcal{M}[\![q']\!]\sigma$. Therefore x *DVD* y in $s \implies x$ *DVD* y in s'.

Conversely, assume x *DVD* y in s' and $y \notin L$. By definition, for some q' in $[s']$, there exist two states σ_1 and σ_2 differing only on y such that $\mathcal{M}[\![q']\!]\sigma_1 x = \mathcal{M}[\![q']\!]\sigma_2 x$. Let the values of the l_i in σ_1 and σ_2 be v_i. By property 6.6.2, we can rewrite the expressions d'_i of q' to give d_i, say, so that they do not depend on l_i but they agree with the values of d_i' for all states where $l_i = v_i$. Call the resulting program q. Clearly $\mathcal{M}[\![q]\!]\sigma_1 x = \mathcal{M}[\![q]\!]\sigma_2 x$. But σ_1 and σ_2 differ only on y and therefore $x\ DVD\ y$ in s.

Clearly the same argument works for $DTVD$ and $DTLD$. This result means that we can always work with variable dependence. Any algorithms we find for variable dependence can be adapted to apply to label dependence.

6.6.3 Variables and Labels Combined

Put another way, there is no need to distinguish between labels and variables and therefore no need to distinguish between *variable data depends* and *label data depends* nor between *variable (terminating) control depends* and *label (terminating) control depends.* For each symbolic value δ, all that is required is the set of names of δ.

Definition 6.6.1 (*names*)
Let δ be as symbolic value

$$names(\delta) = labels(\delta) \cup variables(\delta)$$

Definition 6.6.2 (Data Dependence)

$$datadepends(t)(x) = \bigcup_{\sigma \in range(pfun(t))} names(\sigma\ x)$$

Definition 6.6.3 (Tcontrols)

$$Tcontrols(t)(v) = \bigcup_{\{(\pi,\pi') \mid \perp \neq pfun(t)(\pi)v \neq pfun(t)(\pi')v \neq \perp\}} \left(\bigcap_{\delta \in diffs(\pi,\pi')} names\ \delta \right)$$

Definition 6.6.4 (controls)

$$controls(t)(v) = \bigcup_{\{(\pi,\pi')|pfun(t)(\pi)v \neq pfun(t)(\pi')v\}} \left(\bigcap_{\delta \in diffs(\pi,\pi')} names\ \delta \right)$$

Definition 6.6.5 (DTslice)

Let t be a Symbolic Execution Tree. The DTslice for variable v in t is the union of data dependence and terminating control dependence of v. i.e.

$$\text{DTslice}(t)(v) = datadepends(t)(v) \cup Tcontrols(t)(v)$$

Definition 6.6.6 (Dslice)

Let t be a Symbolic Execution Tree. The Dslice for variable v in t is the union of data dependence and control dependence of v. i.e.

$$\text{Dslice}(t)(v) = datadepends(t)(v) \cup controls(t)(v)$$

On its own, the DTslice computes DTD, the union of $DTVD$ and $DTLD$

Definition 6.6.7 (DTD)

Let s be a schema, then $x\ DTD\ y$ in $s \iff x\ DTLD\ y$ in s or $x\ DTVD\ y$ in s.

and the Dslice computes DD, the union of DVD and DLD

Definition 6.6.8 (DD)

Let s be a schema, then $x\ DD\ y$ in $s \iff x\ DLD\ y$ in s or $x\ DVD\ y$ in s.

6.6.4 Computing $DTVD$ and $DTLD$ using the DTslice

Clearly, the $DTVD$ of the 'label added' schema is the same as the DTD of the original schema.

The $DTVD$ of a schema s can thus be computed in three different ways:-

- It can be computed using the DTVslice as described in Section 6.4.
- It can be computed using the DTslice and then restricting the range of the result to just variables.

- It can be computed by computing the DTVslice of the 'label added' schema of s and then restricting the range of the result to just variables.

Similarly, to compute $DTLD$, we can simply restrict the range of the DTD to just labels and similarly, DVD and DLD can be computed using the Dslice .

6.7 Implementation of $DTLD$ for Loop–free Schemas

In this section we give the algorithm for $DTLD$ in the functional programming language Hope. This program refers to the functions which implemented the semantics of schemas given in Section 4.6.

6.7.1 *labels* (Definition 6.2.2(page 173))

```
labels: delta ->  set name;
labelsset: (set delta) -> (set name);

labels (va x) <= empty;
labels (complex (f,S))   <=  (singleton f) U labelsset (S);
labels botdelta <= empty;
labelsset S  <=mapset(labels,S);
```

6.7.2 *Ldatadepends* (Definition 6.2.1(page 173))

```
datadependent:  (pfun path delta) -> set name ;
datadependent  <= labelsset  o range;
```

6.7.3 *diffs* (Section 5.2.4)

```
differences: path X path -> set delta;
differences((p1,p1'),(p2,p2')) <= ((p1 intersect p2') U (p1' intersect p2));
```

6.7.4 *LTcontrols* (Section 6.2.3)

```
allintersect: set delta  -> set name;
allintersect S <= if S = empty
                  then empty
                  else let (a,T)==choose S
                       in if (card S) = 1
                          then  (labels a)
                          else (labels a) intersect (allintersect T);
```

```
controldependent :  (pfun path delta) -> set name ;
controldependent f <=
mapset(lambda d1 => mapset(
lambda d2 =>
if (apply f d1) = (apply f d2) or (apply f d1)=botdelta  or (apply f d2)=botdelta
then empty
else allintersect (differences(d1,d2)) , domain f),
                            domain f) ;
```

6.7.5 DTLslice (Definition 6.2.4(page 182))

```
dependent:  (pfun path delta) -> (set name);
dependent f <= (datadependent f) U  (controldependent f);
```

6.8 Implementation of *DTVD* for Loop–free Schemas

In this section we give the algorithm for *DTVD* in the functional programming language Hope. This program refers to the functions which implemented the semantics of schemas given in Section 4.6.

6.8.1 *variables* (Definition 6.4.2(page 194))

```
variables: delta ->  set name;
variablesset: (set delta) -> (set name);

variables (va x) <= x & empty;
variables (complex (f,S))   <=  variablesset (S);
variables botdelta <= empty;

variablesset S  <=mapset(variables,S);
```

6.8.2 *Vdatadepends* (Definition 6.4.1(page 194))

```
Vdatadependent:  (pfun path delta) -> set name ;
Vdatadependent  <= variablesset  o range;
```

6.8.3 *VTcontrols* (Definition 6.4.3(page 195))

```
allintersect: set delta  -> set name;
allintersect S <= if S = empty
                  then empty
                  else let (a,T)==choose S
                       in if (card S) = 1
```

```
                                then  (variables a)
                                else (variables a) intersect (allintersect T);


VTcontroldependent :  (pfun path delta) -> set name ;
VTcontroldependent f <=
mapset(lambda d1 => mapset(
lambda d2 =>
if (apply f d1) = (apply f d2) or (apply f d1)=botdelta  or (apply f d2)=botdelta
then empty
else allintersect (differences(d1,d2)), domain f)
                                ,domain f) ;
```

6.8.4 DTVslice (Definition 6.4.4(page 195))

```
DTVslice:  (pfun path delta) -> (set name);
DTVslice f <= (Vdatadependent f) U  (VTcontroldependent f);
```

6.9 Implementation of DTD for Loop–free Schemas

In this section we give the algorithm for DTD in the functional programming language Hope. DTD combines $DTLD$ and $DTVD$.

```
labelsset S  <=mapset(labels,S);

names: delta ->  set name;
names(x) <= (labels x) U (variables x);

nameset: (set delta) -> (set name);
nameset S  <= mapset(names,S);
```

6.9.1 Data depends (Definition 6.6.2(page 200))

```
datadependent:  (pfun path delta) -> set name ;
datadependent  <= nameset o range;


allintersect: set delta  -> set name;
allintersect S <=
if S = empty
then empty
else let (a,T)==choose S
     in if (card S) = 1
```

```
        then  (names a)
        else (names a) intersect (allintersect T);
```

6.9.2 *Tcontrols* (Definition 6.6.3(page 200))

```
Tcontroldependent :  (pfun path delta) -> set name ;
Tcontroldependent f <=
mapset(lambda d1 => mapset(
lambda d2 =>
if (apply f d1) = (apply f d2) or (apply f d1)=botdelta  or (apply f d2)=botdelta
then empty
else allintersect (differences(d1,d2))
                                                      , domain f)
                           ,domain f) ;
```

6.9.3 DTslice (Definition 6.6.5(page 201))

```
DTslice:  (pfun path delta) -> (set name);
DTslice f <= (datadependent f) U  (Tcontroldependent f);
```

6.10 Conclusion

We have formally stated and proved the *DTLD* and *DTVD* algorithms correct for loop free schemas.

From now on, we take advantage of the simplification discussed in Section 6.6, which allows us to treat labels as variables.

DTD was defined as the union of *DTVD* and *DTLD* and hence either of the smaller relations can be defined by simply restricting the range *DTD* to variables and labels respectively.

In the next chapter we complete the work by extending the algorithms to handle schemas that are not necessarily loop free.

Chapter 7

Computing Dataflow Dependencies of Schemas with Loops

7.1 Introduction

In this chapter, the algorithms introduced in Chapter 6, which compute dataflow dependencies of loop–free schemas, are extended to compute the $DTVD$[1] of schemas that contain loops.

The algorithm works by *unfolding* all the loops within a schema. A schema, which has had all its loops replaced by an unfolding is loop–free and hence $DTVD$ can be computed using the techniques described in Chapter 6.

It is proved that if a schema containing loops is unfolded *sufficiently*[2], the resulting loop–free schema will have the same $DTVD$ as the original schema with loops from which it was derived.

The problem of computing $DTVD$ has thus been reduced to the problem of recognising when a schema with loops has been sufficiently unfolded.

At the time of writing, unfortunately, we are not certain how to recognise when this *maximal* number of unfoldings has been reached. There are three possibilities.

Possibility 1 A 'maximal unfolding number[3]' of a schema can be computed recursively from 'crude' information about the structure of its abstract syntax tree.

[1] and hence, the $DTLD$

[2] A finite number of times.

[3] It need not be the least. *Any* one will be sufficient

Hausler [52] proved[4] that for every program p, a maximal number of unfoldings of p that are necessary to capture all dependence information is computable. His algorithm computes the maximal unfolding number of each compound syntactic construct in a program recursively in terms of the maximal unfolding numbers of each of its components. The maximum unfolding number for each basic syntactic construct is constant (For example, *one* for an assignment statement).

Possibility 2 There is some relationship between schemas such that if, two successive iterations of the unfolding process, give rise to schemas which are related in this way, then further iterations cannot introduce new dependencies.

Possibility 3 Recognising when a schema has been maximally unfolded is not computable.

For the dataflow dependencies introduced in this thesis, it seems very likely that the first of these is true. For example, in the case of a 'tiny' loop consisting of a single assignment statement, it can be seen that after two unfoldings all the dependence information has been gathered and further unfolding cannot add any new dependencies. As in the case of Hausler's work, it is highly likely that the maximal unfolding number for each syntactic construct in the language of schemas can also be expressed as a function of the maximal unfolding numbers of each of its components.

Taking advantage of this, an algorithm for $DTVD$ would compute a maximal unfolding number n_i for each syntactic component s_i of the schema and then simply unfold each component s_i, n_i times. The methods introduced in Chapter 6 could then be used to compute the $DTVD$ of the resulting loop–free schema.

Since at the time of writing, we do not know how to compute the maximal unfolding numbers for[5] dataflow dependencies, we cannot use them in our algorithms. The algorithms that we have implemented rely on the second assumption being true. The relationship that we use is that the two schemas have the same $DTD (= DTVD \cup DTLD)$[6]. As is demonstrated, unfolding is monotonic, i.e. further unfolding cannot reduce the $DTVD$. It also bounded

[4] We cannot necessarily assume that his result is true in our case, since we are computing different dependencies from Hausler.

[5] although, we believe them to be the same as Hausler's.

[6] There are probably others. Perhaps, for example, data dependence on its own is sufficient. It certainly works in all the examples that have been tested and is much more efficient.

above by a finite object. *If* it had been proved that no change in the *DTD* in *one* iteration of the unfolding process implied that further iterations *could not* introduce further changes to the *DTVD*, then the algorithm *would have been* proved correct.

The first and third possibilities are mutually contradictory. If the ability to recognise when a loop is maximally unfolded were not, in general decidable, it would mean there could be no connection between the 'size' of a loop and the number of iterations of it that were required before all dependence information was 'gathered'. It would mean for some classes of programs, no such maximum upper limit based on crude syntactic properties involving the numbers of statements would exist. In view of Hausler's work, this seems highly unlikely.

7.2 Unfoldings

Hausler [52] defines the denotational slice of a while loop in terms of unfolding the loop as a nested conditional. A very similar approach is used in this thesis.

Definition 7.2.1 (Unfolding)
Given a schema $[\![\textit{while } b \textit{ do } S]\!]$, define the sequence of schemas:

$W_0(b, S) = \textit{if } b \textit{ then FAIL else skip}$
$W_{n+1}(b, S) = \textit{if } b \textit{ then } S; W_n(b, S) \textit{ else skip.}$

We call $W_i(b, S)$ the *ith unfolding* of $[\![\textit{while } b \textit{ do } S]\!]$.

7.2.1 Example of unfolding

Consider the schema:-

```
while b1(x, y)
do if b2(x)
    then x := f(x, y)
    else y := g(y)
```

The zeroth unfolding: W_0=

```
if b1(x, y)
then FAIL
else skip
```

The first unfolding is given by : $W_1 =$

```
if b1(x, y)
then S; W0
else skip
```

where S is the body of the loop i.e. $W_1 =$

```
if b1(x, y)
then if b2(x)
        then x := f(x, y)
        else y := g(y);
        W0
else skip
```

i.e. $W_1 =$

```
if b1(x, y)
then   if b2(x)
          then x := f(x, y)
          else y := g(y);
          ┌──────────────┐
          │ if b1(x, y)  │
          │ then FAIL    │
          │ else skip    │
          └──────────────┘
else skip
```

The second unfolding, $W_2 =$

```
if b1(x, y)
then if b2(x)
        then x := f(x, y)
        else y := g(y);
        W1
else skip
```

The second unfolding, W_2 is thus

```
if b1(x, y)
then  if b2(x)
      then x := f(x, y)
      else y := g(y);
          if b1(x, y)
          then   if b2(x)
                 then x := f(x, y)
                 else y := g(y);
                   if b1(x, y)
                   then FAIL
                   else skip
          else skip
else skip
```

Observation 7.2.1 *To get from $W_n(b, S)$ to $W_{n+1}(b, S)$, the occurrence of FAIL is replaced by*

$$[\![S; \textit{if } b \textit{ then FAIL else skip}]\!]$$

7.3 The *DTVD* Algorithm for Loop Schemas

The way the algorithm works with loops is that loops are repeatedly unfolded. Eventually a stage will be reached where the *DTVD* is *maximal* in the sense that further unfoldings will not cause further changes in the *DTVD*.

7.3.1 Example

Consider, again, the schema,

```
while b1(x, y)
do if b2(x)
   then x := f(x, y)
   else y := g(y)
```

. Using the semantics of loop free schemas given in Chapter 4, the zeroth unfolding, W_0=

```
if b1(x, y)
then FAIL
else skip
```

has symbolic execution tree $\mathcal{S}[\![W_0]\!]$ given in Figure 7.1(page 212).

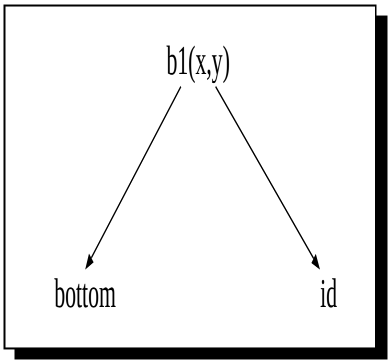


Figure 7.1: $\mathcal{S}[\![W_0]\!]$: The symbolic execution tree of W_0

The data dependence DTslice (Definition 6.6.5(page 201)) is computed The results are shown in Figure 7.2(page 212): The variables x and y are not dataflow terminating dependent on any variable or label[7].

Variable	Data	Control
x	$\emptyset$	$\emptyset$
y	$\emptyset$	$\emptyset$

Figure 7.2: DTslice of $\mathcal{S}[\![W_0]\!]$

[7] If we had used non-terminating control dependence (Definition 6.6.4(page 200)), x and y would both have depended on x, y and b_1.

The loop is unfolded once more to give $W_1 =$

```
if b1(x, y)
then if b2(x)
        then x := f(x, y)
        else y := g(y);

        if b1(x, y)
        then FAIL
        else skip
else skip
```

W_1 has the symbolic execution tree $\mathcal{S}[\![W_1]\!]$ given in Figure 7.3(page 213).

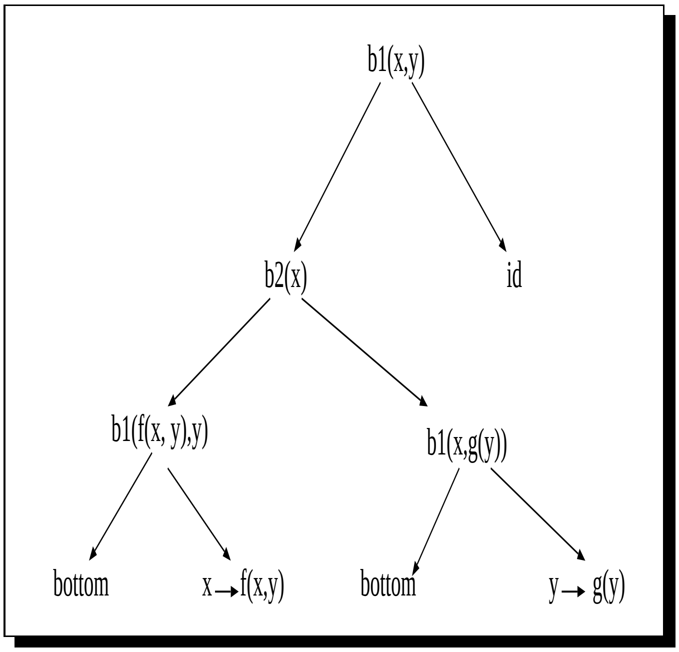


Figure 7.3: $\mathcal{S}[\![W_1]\!]$: The symbolic execution tree of W_1

Again, the DTslice is computed. The results are shown in Figure 7.4(page 213).

Variable	Control	Data
x	$\{b_1, b_2, x, y\}$	$\{f, x, y\}$
y	$\{b_1, b_2, x, y\}$	$\{g, y\}$

Figure 7.4: DTslice of $\mathcal{S}[\![W_1]\!]$

The second unfolding, $W_2 =$

```
if b1(x, y)
then if b2(x)
     then x := f(x, y)
     else y := g(y);
     ┌────────────────────────
     │ if b1(x, y)
     │ then if b2(x)
     │      then x := f(x, y)
     │      else y := g(y);
     │       if b1(x, y)
     │       then FAIL
     │       else skip
     │ else skip
     └────────────────────────
else skip
```

has the symbolic execution tree $\mathcal{S}[\![W_2]\!]$ given in Figure 7.5(page 214).

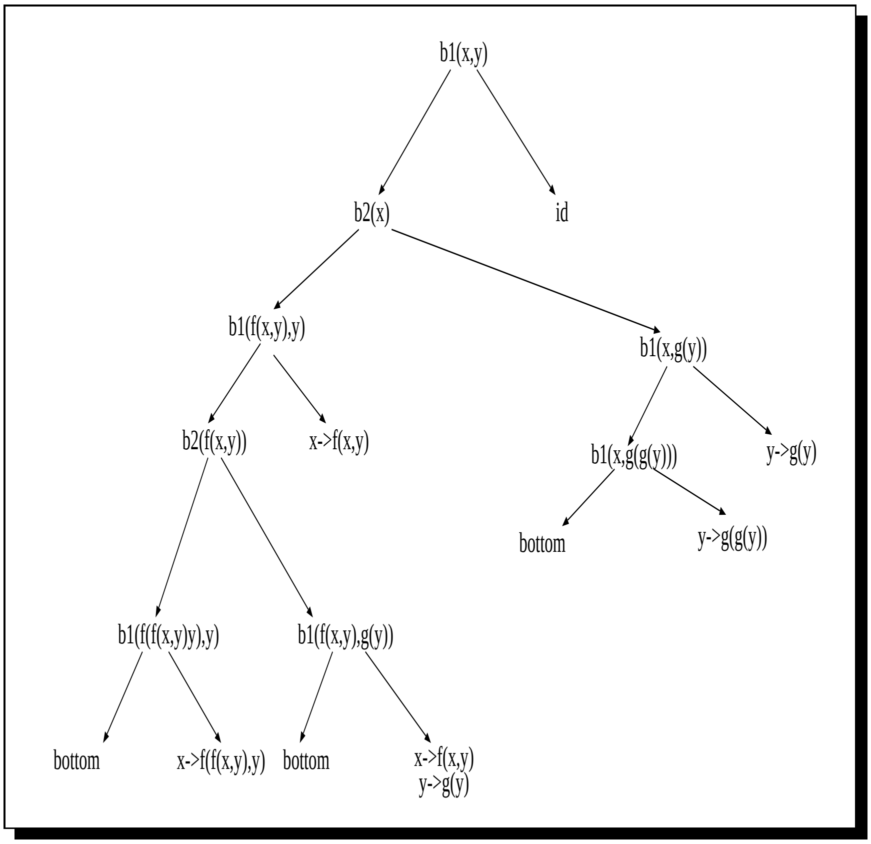


Figure 7.5: The symbolic execution tree, $\mathcal{S}[\![W_2]\!]$ of W_2

Notice, by Observation 7.2.1(page 211), to get from one unfolding to the next we replace each $\perp$ by the symbolic execution tree, $\mathcal{S}[\![S;$ *if* $b1(x, y)$ *then FAIL else skip*$]\!]$ namely, the symbolic execution tree given in Figure 7.6(page 215), evaluated in the state immediately to the right of the $\perp$ being replaced.

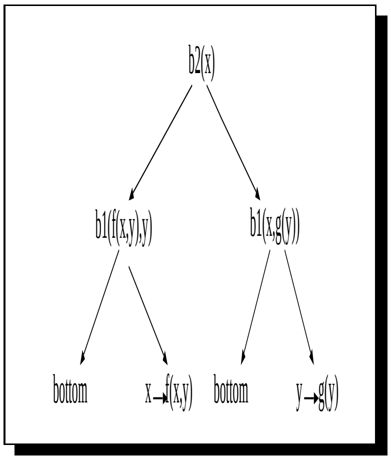


Figure 7.6: $\mathcal{S}[\![S; \mathit{if}\, b1(x,y)\ \mathit{then}\ \mathit{FAIL}\ \mathit{else}\ \mathit{skip}]\!]$

Notice also, that at this stage, some simplification (Section 4.4.4) has taken place. The symbolic execution tree corresponding to $\mathcal{S}[\![W_2]\!]$ before pruning is given in Figure 7.7(page 216).

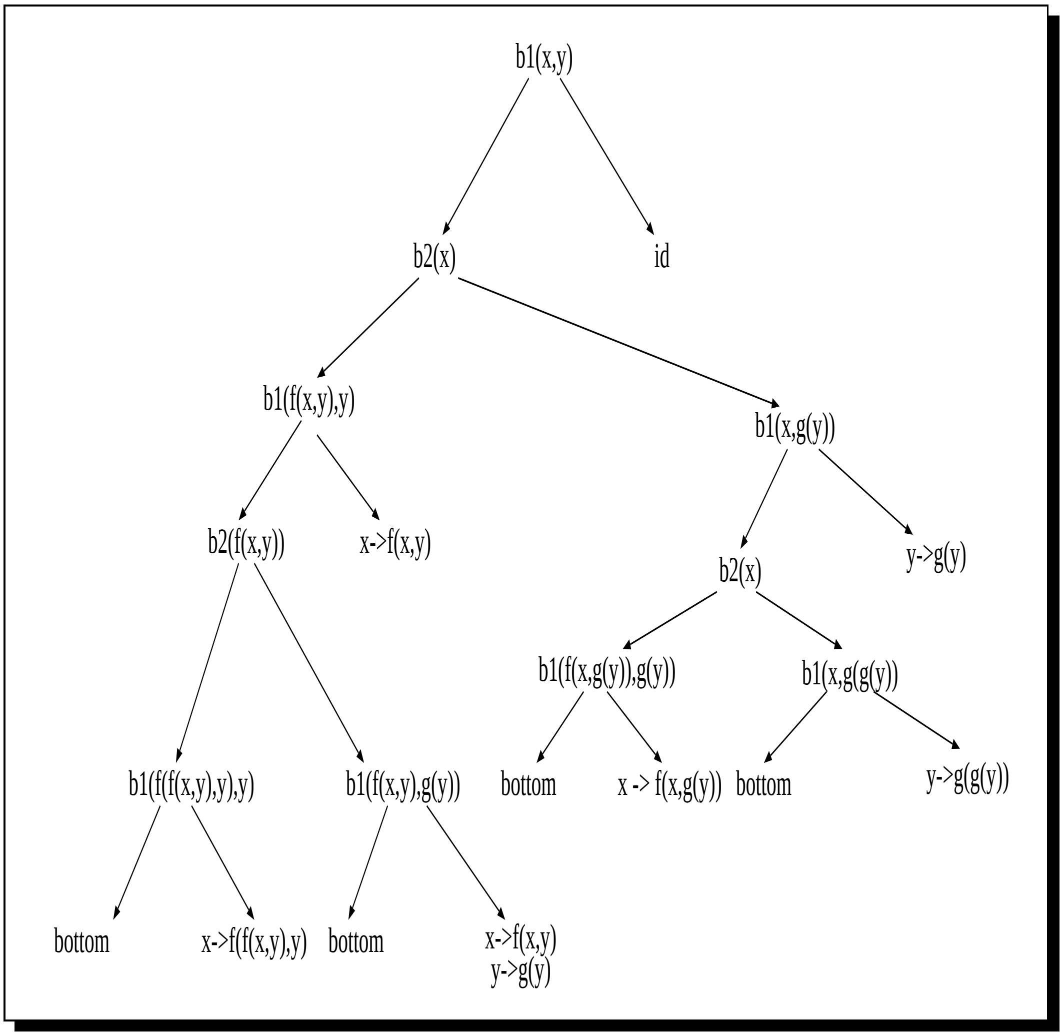


Figure 7.7: The symbolic execution tree, $\mathcal{S}[\![W_2]\!]$ before pruning

The subtree to the left of the lower occurrence of $b_2(x)$ has been removed since it is represents impossible paths.

Again, this time using $\mathcal{S}[\![W_2]\!]$, the DTslice is computed The results are shown in Figure 7.8(page 216).

Variable	Control	Data
x	$\{b_1, b_2, f, x, y\}$	$\{f, x, y\}$
y	$\{b_1, b_2, f, x, y\}$	$\{g, y\}$

Figure 7.8: DTslice of $\mathcal{S}[\![W_2]\!]$

A new control dependence of x on f has emerged.

The process is repeated once more. W_3 is the schema which has the symbolic execution tree $\mathcal{S}[\![W_3]\!]$ given in Figure 7.9(page 217).

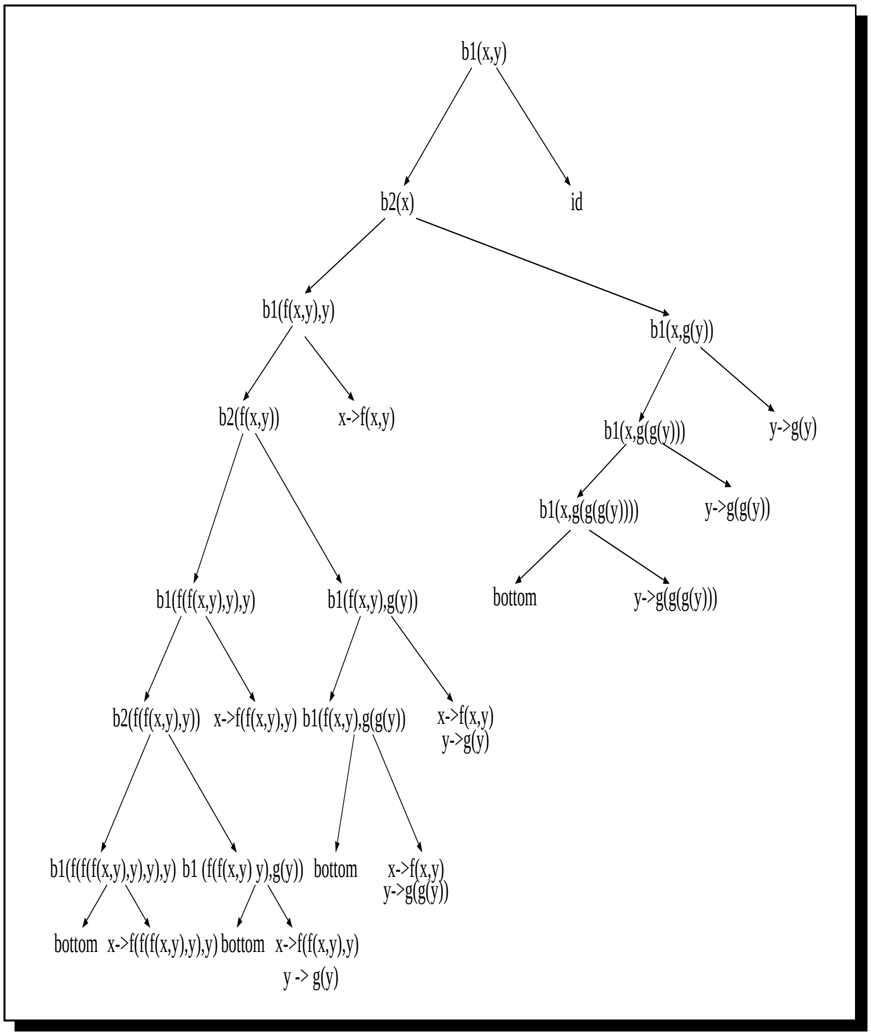


Figure 7.9: The Symbolic Execution Tree, $\mathcal{S}[\![W_3]\!]$

This time we find there is no change in the DTslice (Figure 7.10(page 217)). The schema has reached its maximal unfolding so the algorithm terminates.

Variable	Control	Data
x	$\{b_1, b_2, f, x, y\}$	$\{f, x, y\}$
y	$\{b_1, b_2, f, x, y\}$	$\{g, y\}$

Figure 7.10: DTslice of $\mathcal{S}[\![W_3]\!]$

Since it is loop–free, to produce the *DTVD* for each variable(see Figure 7.11(page 218)), we simply compute the DTVslice (Definition 6.4.4(page 195)) of this maximal unfolding ($\mathcal{S}[\![W_3]\!]$) (or simply restrict the range of the DTslice to just variables as described in Chapter 6).

Variable	$DTVD$
x	$\{x, y\}$
y	$\{x, y\}$

Figure 7.11: $DTVD$ of the Loop Schema

and to produce the $DTLD$ for each variable(see Figure 7.12(page 218)), we simply compute the DTLslice (Definition 6.2.4(page 182)) of $\mathcal{S}[\![W_3]\!]$ (or simply restrict the range of the DTslice to just labels as described in Chapter 6).

Variable	$DTLD$
x	$\{b_1, b_2, f\}$
y	$\{b_1, b_2, f, g\}$

Figure 7.12: $DTLD$ of the Loop Schema

Notice that the DTLslice with respect to x does not contain g.

7.4 Dataflow Dependence of Unfoldings

The relationship between a loop schema and its ith unfolding is embodied in the following lemma. Lemma 7.4.1 states that a program represented by a loop schema and the corresponding program in the ith unfolding of the loop schema agree in all states where the loop terminates in i or fewer iterations. In all other states the latter program will not terminate.

Lemma 7.4.1 *Let $p \in [\textit{while } b \textit{ do } S]$ and let p' be the 'corresponding' program in $[W_i(b, S)]$. Then for all states σ in which p terminates in i or fewer iterations*

$$\mathcal{M}[\![p]\!]\sigma = \mathcal{M}[\![p']\!]\sigma$$

and in all other states $\mathcal{M}[\![p']\!]\sigma = \perp$.

Corollary 7.4.1 *Let $[\![\textit{while } b \textit{ do } S]\!]$ be a loop schema. Let i and j be natural numbers with $i > j$. Let p_i and p_j be 'corresponding' programs in its ith unfolding,$[W_i(b, S)]$ and jth* *unfolding,*$[W_j(b, S)]$ *respectively. Then for all states* σ *in which* p_i *terminates in* j *or fewer iterations*

$$\mathcal{M}[\![p_i]\!]\sigma = \mathcal{M}[\![p_j]\!]\sigma$$

and in all other states $\mathcal{M}[\![p_j]\!]\sigma = \perp$.

7.4.1 A Partial Ordering on Programs

We use the standard ordering on programs defined in denotational semantics [78, 88].

Definition 7.4.1

Given programs p_1 and p_2,

$$p_1 \sqsubseteq p_2 \iff \mathcal{M}[\![p_1]\!] \sqsubseteq \mathcal{M}[\![p_2]\!]$$

where the ordering on program meanings is defined as follows:-

Definition 7.4.2

$$\mathcal{M}[\![p_1]\!] \sqsubseteq \mathcal{M}[\![p_2]\!] \iff \text{for all states } \sigma, \mathcal{M}[\![p_1]\!]\sigma \sqsubseteq \mathcal{M}[\![p_2]\!]\sigma$$

where the ordering on states is defined as follows:-

Definition 7.4.3

$$\sigma_1 \sqsubseteq \sigma_2 \iff (\sigma_1 = \sigma_2) \text{ or } (\sigma_1 = \perp)$$

Using this ordering, program $p_1 \sqsubseteq p_2$ if and only if whenever p_1 terminates, p_2 terminates in the same state.

Using the set theoretic definition of a binary relation it can be seen that:-

Lemma 7.4.2

$$p_1 \sqsubseteq p_2 \implies TVD(p_1) \subseteq TVD(p_2)$$

Definition 7.4.4 (Program Substitution)

Let p be a program containing sub–program p'.

$p[p''/p']$ is the program p with p' replaced by p''.

Lemma 7.4.3 (Program Substitution Lemma)

$$p' \sqsubseteq p'' \implies p[p'/q] \sqsubseteq p[p''/q]$$

Proof: It follows straightforwardly from the continuity [78] and strictness of all the operators in standard semantics.

Corollary 7.4.2

$$p' \sqsubseteq p'' \implies TVD(p[p'/q]) \subseteq TVD(p[p''/q])$$

Proof: : Follows immediately from Lemma 7.4.2(page 219).

7.4.2 A Partial Ordering on Schemas

The partial ordering on programs defined in Definition 7.4.1(page 219) gives rise to a partial order on schemas.

Definition 7.4.5

$$s_1 \sqsubseteq s_2$$

if and only if for all 'corresponding' pairs of programs (p_1, p_2) in $[s_1] \times [s_2]$,

$$p_1 \sqsubseteq p_2$$

Corollary 7.4.3

$$s_1 \sqsubseteq s_2 \implies DTVD(s_1) \subseteq DTVD(s_2)$$

Proof: : Follows immediately from Lemma 7.4.2(page 219).

Definition 7.4.6 (Schema Substitution)

Let s be a schema containing sub–schema s'.

$s[s''/s']$ is the program s with s' replaced by s''.

Lemma 7.4.4 (Schema Substitution Lemma)

$$s' \sqsubseteq s'' \implies s[s'/t] \sqsubseteq s[s''/t]$$

Proof: : Follows immediately from Lemma 7.4.3(page 220).

Corollary 7.4.4

$$s' \sqsubseteq s'' \implies DTVD(s[s'/t]) \subseteq DTVD(s[s''/t])$$

Proof: : Follows immediately from Lemma 7.4.4(page 221).

Corollary 7.4.5 *Let $[\![while\ b\ do\ S]\!]$ be a loop schema. Let i and j be natural numbers with $i > j$. Then*

$$W_j(b,S) \sqsubseteq W_i(b,S) \sqsubseteq [\![while\ b\ do\ S]\!]$$

Proof: : Follows immediately from Observation 7.2.1(page 211),Corollary 7.4.4(page 221), and Lemma 7.4.1(page 218).

Lemma 7.4.5 *Let s be a schema that mentions the set of variables V and no others. Then*

$$DTVD(s) \subseteq id \cup (V \times V)$$

Proof: obvious

Lemma 7.4.6 *For all loops schemas, $[\![while\ b\ do\ S]\!]$,*

$$n < m \implies DTVD(W_n(b,S)) \subseteq DTVD(W_m(b,S))$$

Proof: Follows immediately from Corollary 7.4.5(page 221).

Lemma 7.4.7 *For all loops schemas, $[\![while\ b\ do\ S]\!]$, for all natural numbers n,*

$$DTVD(W_n(b,S)) \subseteq DTVD([\![while\ b\ do\ S]\!])$$

Proof: Follows immediately from Corollary 7.4.5(page 221) and Corollary 7.4.3(page 220).

Corollary 7.4.6 *For all loops schemas, $[\![while\ b\ do\ S]\!]$, there exists a natural number n such that*

$$m > n \implies DTVD(W_m(b, S)) = DTVD(W_n(b, S))$$

Proof: Follows immediately from the fact $DTVD$ is monotonic(Corollary 7.4.3(page 220)) and 'sandwiched' between id and $id \cup (V \times V)$. Where V is the finite set of variables mentioned in $[\![while\ b\ do\ S]\!]$.

Definition 7.4.7 (Maximal Unfolding Number)
Given a loop schema, $[\![while\ b\ do\ S]\!]$, we define the *least* natural number n such that

$$m > n \implies DTVD(W_m(b, S)) = DTVD(W_n(b, S)).$$

as the *maximal unfolding number* of $[\![while\ b\ do\ S]\!]$.

Definition 7.4.8 (Maximal Unfolding)
We call $W_N(b, S)$ the maximal unfolding of $[\![while\ b\ do\ S]\!]$

We now prove that for all while loop schemas W, there exists an N such that W and its Nth unfolding, W_N have the same $DTVD$.

Theorem 7.4.1 *For all while loop schemas, $[\![while\ b\ do\ S]\!]$,*

$$DTVD[\![while\ b\ do\ S]\!] = DTVD(W_N(b, S))$$

where N is the maximal unfolding number of $[\![while\ b\ do\ S]\!]$.

Proof: If $x\ DTVD\ y$ in $[\![while\ b\ do\ S]\!]$ there must exist two states σ_1 and σ_2 differing only on y such that $[\![while\ b'\ do\ S']\!]$ terminates with different final values for x where $[\![while\ b'\ do\ S']\!]$ is a program represented by $[\![while\ b\ do\ S]\!]$. Since $[\![while\ b'\ do\ S']\!]$ terminates there must be some n such that it terminates after n iterations of the loop. There must therefore exist some program represented by its nth unfolding, $W_n(b, S)$, that behaves the same as $[\![while\ b'\ do\ S']\!]$ in states σ_1 and σ_2, i.e. $x\ DTVD\ y$ in $W_n(b, S)$. But if $x\ DTVD\ y$ in $W_n(b, S)$ then $x\ DTVD$

y in $W_N(b,S)$ where N is the maximal unfolding number of $[\![while\ b'\ do\ S']\!]$. We have shown that x $DTVD$ y in i.e.

$$[\![DTVD(while\ b\ do\ S]\!]) \subseteq DTVD(W_N(b,S))$$

but Lemma 7.4.7(page 221),

$$DTVD(W_N(b,S)) \subseteq DTVD([\![while\ b\ do\ S]\!])$$

and therefore

$$DTVD(W_N(b,S)) = DTVD[\![while\ b\ do\ S]\!]$$

as required.

We have proved that the process of unfolding a while loop schema will eventually produce a loop–free schema whose $DTVD$ is the same as that of the original while loop schema.

Starting with a schema s containing many possibly nested while loops, consider an algorithm that works as follows:-

Algorithm 7.4.1

1. Unfold every loop in s once.

2. repeat
   unfold each loop one more time
   forever

Lemma 7.4.8 *After a finite number of iterations of Algorithm 7.4.1 above, the schema produced by Algorithm 7.4.1 will reach a point, s_{max}, where further unfoldings will never change its DTVD. We call the schema s_{max}, the Maximal Unfolding of s.*

Proof: By Corollary 7.4.5(page 221) and Corollary 7.4.4(page 221), each iteration will produce a schema with a bigger $DTVD$ and it will never be bigger than the original schema s. By Lemma 7.4.5(page 221), the $DTVD$ of s is bounded above by a finite relation.

Lemma 7.4.9 *Algorithm 7.4.1 must lead to a schema whose DTVD is the same as the original schema s.*

Proof: From schema substitution lemma(Lemma 7.4.4(page 221)) and Lemma 7.4.7(page 221), it follows that the $DTVD$ of each unfolded schema described above will always be a subset of the original schema s. Therefore

$$DTVD(s_{max}) \subseteq DTVD(s)$$

Conversely, and again similar to the proof in Theorem 7.4.1(page 222), suppose x $DTVD$ y in s. Then there exists a program p in $[s]$ and two states,σ_1 and σ_2 differing only at y in which p terminates with different final values for x. In the execution of p to reach this final state, for each loop l in p there will be maximum number n_l, say, of times it got executed (in states σ_1 and σ_2) . Then by Corollary 7.4.1(page 218),Corollary 7.4.3(page 220) and the program substitution lemma((Lemma 7.4.3(page 220)), if we replace each loop l by its n_lth unfolding this program will behave the same as p in both states σ_1 and σ_2. But by Lemma 7.4.8(page 223), the schema t representing this program will be such that

$$DTVD(t) \subseteq DTVD(s_{max})$$

Therefore x $DTVD$ y in s_{max}

$$DTVD(s) \subseteq DTVD(s_{max})$$

Hence

$$DTVD(s) = DTVD(s_{max})$$

as required.

We have shown that *sufficient* unfoldings of a schema s containing loops would result in a loop free schema whose $DTVD$ is the same as s. Provided an algorithm performs sufficient unfoldings it will be correct.

7.5 Implementation of *DTVD* and *DTLD* for Schemas with Loops

This algorithm works by repeatedly unfolding innermost loops and computing *DTD* using the algorithms defined in Chapter 6 until there is no change in the *DTD* of the resulting unfolding. It works outwards until the schema is loop free. When this situation is reached the *DTLD* and *DTVD* can be computed by restricting the range of *DTD* to just variables and just labels respectively as described in Section 6.6.

7.5.1 The Set of Variables Affected by a Schema

```
affected: statement -> set variable;
affectedl: (list statement) -> set variable;
affected bottom <= empty;
affected (ass(x,E)) <= singleton x;
affected (ife(E,s1,s2)) <= (affectedl s1) U (affectedl s2);
affected (while(E,s)) <= affectedl s;
affectedl nil <= empty;
affectedl (x::l) <= (affected x) U (affectedl l);
```

7.5.2 A Function which checks whether two symbolic execution trees have the same DTslice

```
samedependency: (set variable)->SET->SET->bool;
samedependency A t1 t2<=
makepfun A (DTslice o  (pathfun t1))=makepfun A (DTslice o (pathfun  t2));
```

7.5.3 Implementation of Unfolding

```
least: (set variable) -> delta -> (list statement) -> statement -> SET;
least A b l s <=  let (t1,t2) == (meaning s, meaning (ife(b,l<>[s],[])))
                  in if samedependency A t1 t2
                     then t2
                     else least A b l (ife(b,l<>[s],[]));

meaning (while(b,l))  <=
least (affected (while (b,l))) b l (ife(b,[FAIL],[]));
```

7.5.4 Implementation of DTD

```
DTLD: list statement -> pfun variable (set name);
DTD l <=
makepfun (affectedl l)
(DTslice o (pathfun  (meaningl l)));
```

7.6 Implementation of $DTLD$

We simply restrict the range of DTD to the set of labels that occur in the schema.

```
alllabels: statement -> set name;
alllabelsl: (list statement) -> set name;
alllabels bottom <= empty;
alllabels (ass(x,E)) <=  (labels E);
alllabels (ife(E,s1,s2)) <= (labels E) U (alllabelsl s1) U (alllabelsl s2);
alllabels (while(E,s)) <= (labels E) U (alllabelsl s);
alllabelsl nil <= empty;
alllabelsl (x::l) <= (alllabels x) U (alllabelsl l);

sortofrangerestrict: set name ->  pfun name (set name) ->
pfun name (set name);

sortofrangerestrict S f <=
if f=empty
then empty
else  let ((a,b),g) == choose f
      in  (a,b intersect S) & (sortofrangerestrict S g);
```

and then

```
DTLD: list statement -> pfun name (set name);
DTLD s <= sortofrangerestrict (alllabelsl s) (DTD s);
```

7.7 Implementation of $DTVD$

Similarly, we simply restrict the range of DTD to the set of variables that occur in the schema.

```
allvariables: statement -> set name;
allvariablesl: (list statement) -> set name;
allvariables bottom <= empty;
allvariables (ass(x,E)) <= x & (variables E);
allvariables (ife(E,s1,s2)) <= (variables E) U (allvariablesl s1) U (allvariablesl s2);
allvariables (while(E,s)) <= (variables E) U (allvariablesl s);
allvariablesl nil <= empty;
allvariablesl (x::l) <= (allvariables x) U (allvariablesl l);
```

and then

```
DTVD: list statement -> pfun name (set name);
DTVD s <= sortofrangerestrict (allvariablesl s) (DTD s);
```

Chapter 8

Conclusions

In this thesis, we set out to answer the following questions:

1. Why do dataflow slicing algorithms, like Weiser's, produce slices that are not dataflow minimal?

2. Do algorithms for producing dataflow minimal slices exist?

8.1 Why do dataflow slicing algorithms, like Weiser's, produce slices that are not dataflow minimal?

'Traditional' data and control dependence expresses a relationship between the nodes of a program's control flow graph [53]. A node n_1 is data or control dependent on node n_2 implies that there exists *an instance* of the execution of the statement corresponding to node n_2 which 'affects' an instance of execution of the statement corresponding to node n_1. This 'affects' relationship is not transitive. Just because there exists an instance of execution of a node n_1 which affects an instance of execution of node n_2, and an instance of execution of node n_2 which affects the final value of x, it *cannot* be assumed that there is an instance of execution of node n_1 that affects the final value of x. This is seen in the ubiquitous example in Figure 1.2(page 24), where

- an execution instance of node 3 affects an execution instance of node 2

- and an execution instance of node 2 affects an execution instance of node 4

- and an execution instance of node 4 affects the final value of x

but there is *no* execution instance of node 3 that affects the final value of x.

Slicing algorithms, for example, [92, 93, 28, 82, 5], use the *transitive closure* of the union of data and control dependence, which by definition *is* transitive and will, as has just been seen, sometimes result in '*extra*' dependencies. The extra dependencies give rise to the undesirable inclusion of 'unnecessary statements' in the slices produced by these algorithms.

8.2 Do algorithms for producing dataflow minimal slices exist?

In an attempt to answer this question, a form of dataflow minimal slice called *DTVD* is introduced. An algorithm for computing the *DTVD* of loop free program schemas is introduced and proved correct.

The algorithm is extended to handle program schemas with loops by using *repeated unfolding*.

An unfolding is, by definition, a loop free program schema and therefore its dataflow minimal slice can be computed using the correct algorithm mentioned above.

For program schemas, s, containing loops we prove that there exists an integer n, which we call the *maximal unfolding number* for s, where the dataflow minimal slice of s is the same as the *DTVD* of its *mth unfolding* for all $m \geq n$. The problem of computing *DTVD* of s is thus reduced to the problem of finding the maximal unfolding number of s.

We have proved that the process of repeatedly unfolding s will eventually reach the maximal unfolding. A remaining problem is that we do not currently know how to recognise when the maximal unfolding has been reached.

One possibility is to repeatedly unfold s until one 'step' produces no further changes in the *DTVD* of s. We believe, but have not yet proved, that reaching this *stable state* implies that s's maximal unfolding number has been reached.

Alternatively, as in the case of Hausler's work [52], it is quite possible that there is a *maximal unfolding number* for each syntactic construct in the language of schemas which can be expressed as a function of the maximal unfolding numbers of each of its components.

Taking advantage of this, an algorithm for *DTVD* would compute a maximal unfolding number n_i for each syntactic component s_i of the schema and then simply unfold each component s_i, n_i times and then apply the algorithm for loop free schemas.

8.3 The Approach

This thesis introduces four dependence relations

- *DVD* –**Dataflow Variable Dependence**
- *DTVD*–**Dataflow Terminating Variable Dependence**
- *DLD* –**Dataflow Label Dependence**
- *DTLD* –**Dataflow Terminating Label Dependence**

These relations are on on control flow graphs and are all expressible in terms of a standard program semantics. If an algorithm exists for any of these dependencies then it will be dataflow minimal.

These dependencies are compared with slicing (Section 3.13). It is claimed that *DTLD* is a dataflow minimal version of Venkatesh's static backward closure slice [91] and *DLD* is a dataflow minimal version of slices which preserve standard semantics rather than the lazy semantics [21] preserved by slices produced by Weiser's algorithm [92, 93] and the PDG approach [82].

Programs are partitioned into *dataflow equivalence classes*. Each dataflow equivalence class represents the set of all programs with the same control flowgraph. Dataflow equivalence classes are represented as **schemas** [44].

A semantics $\mathcal{S}$ of loop-free schemas (Chapter 4) which maps loop-free schemas into **Symbolic Execution Trees** is defined. $\mathcal{S}$ gives rise to the first stage in the algorithm for computing the dependencies. A theory of Symbolic Execution Trees is developed (Chapter 5) and it is proved that $\mathcal{S}[\![s]\!]$ properly characterises the set of programs represented by the loop–free schema s (Theorem 5.4.1(page 160)).

New forms of **Data and Control Dependence** are defined in terms of symbolic execution trees(Chapter 6). For loop–free schemas s, algorithms for computing *DTLD* and *DTVD* of s in terms of these new forms of data and control dependence of the symbolic execution tree, $\mathcal{S}[\![s]\!]$ are defined and proved correct (Theorem 6.3.5(page 188)). This proof relies on certain assumptions about the 'richness' of the expressions of the programming language being analysed (Assumption 3.4.1) as well as the soundness and completeness of $\mathcal{S}$ (Theorem 5.4.1(page 160)).

Similar algorithms for computing DLD and DVD are described but not proved correct.

It is proved that D(T)LD is just a special case of D(V)LD (Section 6.6).

The dataflow dependencies for schemas with loops is computed by an iterative process (Chapter 7). Initially each loop is replaced by its 'zeroth unfolding' and dataflow dependence of this resulting loop–free schema is computed. The resulting schema is further unfolded and the process is repeated. We formally prove that this process will eventually terminate resulting with a loop–free schema whose dataflow dependence is the same as the program with loops with which we started.

Provided that we can recognise when further unfoldings will produce no further changes in dependency, we have achieved algorithms for computing the various minimal dataflow dependencies introduced in this thesis.

Chapter 9

Future Work

The first priorities for future work are

1. Further investigation into the claim that Hausler's maximal unfolding number is applicable to the dataflow dependencies of schemas.

2. Further investigation of the claim that no change in the DTD in *one* iteration of the unfolding process imply that further iterations *cannot* introduce further changes to the $DTVD$.

The proof of either of these claims would complete the proof of the computability of $DTVD$ of schemas with loops.

It is also possible that simpler conditions exist for guaranteeing no further change in the $DTVD$ as a result of unfolding. Possibilities for further investigation include:-

1. Unfolding until there is no further change in just the data dependency[1].

2. Unfolding until no new 'flattened symbolic states' occur: $\{x \mapsto f(f(x)), y \mapsto g(y,z)\}$ and $\{x \mapsto f(x), y \mapsto g(g(y,z),z)\}$ are two examples of the same flattened state, since the set of variables and labels in the symbolic value corresponding to each variable is the same in both states.

Alternatively, a non–constructive approach to the computability of $DTVD$, i.e. a proof that is not dependent on a particular algorithm may bear fruit.

[1] This approach appears to work in the many examples so far tested.

9.1 Extending the Proofs and Algorithms to *DVD* and *DLD*

The main proofs in this thesis have all been for the terminating forms of dataflow dependence: *DTVD* and *DTLD*. In Section 6.5 on page 195 it is claimed that only small changes to the definitions of control dependence are needed to achieve the non–terminating forms *DVD* and *DLD*. Further investigation of these claims is required.

9.2 Improving Efficiency

In this thesis, the question of the existence of an algorithm for computing dataflow minimal slices was posed. The algorithms introduced here sometimes produces slices that are thinner than those produced using current approaches. Since these algorithms examine all paths through a schema they have exponential complexity and are therefore do not scale up to 'real world' problems.

Further work is required to investigate whether this exponential nature is a function of the problem itself or just a function of the particular approach we have used.

9.3 Experimenting with Different Definitions of Control Dependence to Obtain different dependences

It appears that many useful forms of dataflow dependence can be computed by applying subtle changes to the definition of control dependence. For example, the only difference in the computation of *DTVD* and *DVD* is due to such a difference. It was recently noticed[2] that that we can have a schema s such that $\neg(xDTVDy)$ and $\neg(xDTVDz)$ but there exist two states differing only on x and y with different non–terminating values for z. This means, that surprisingly, the initial values of a set of variables can jointly contribute to the final value of a variable even if they do not contribute individually.

This leads us to require a different form of dependency where z is considered to depend both on x and y. This form of dependency is closer to slicing since in a slice we would wish to include all sets of statements who jointly contribute to the final value of a variable even

[2] Thanks to John Howroyd.

if they do not contribute individually. In this form of dependency, which we call, **terminating variable set dependency** *TVSD* and its dataflow companion **dataflow terminating variable set dependency** (*DTVSD*), a variable depends upon a *set* of variables. Note that we do not need **dataflow variable set dependency**(*DVSD*), since *DVSD*=*DVD*.

Definition 9.3.1 (*TVSD*)

Let S be a set of variables. x *TVSDS* in p means there exist two states differing only on S such that

$$\perp \neq \mathcal{M}[\![p]\!]\sigma \ x \neq \mathcal{M}[\![p]\!]\sigma' \ x \neq \perp$$

and for all proper subsets T of S, for all states σ, σ' differing only on T

$$\perp \neq \mathcal{M}[\![p]\!]\sigma \ x = \mathcal{M}[\![p]\!]\sigma' \ x \neq \perp$$

The author believes that the algorithm to produce *DTVSD* can be obtained by a small change in the definition of control dependency. Generally, more research is required into how subtle changes in the definition of control dependency give rise to algorithms for solving different dataflow dependencies.

9.4 Further Applicability of Symbolic Execution Trees

In this section, we briefly consider other possible applications of Symbolic Execution Trees.

The main body of this thesis has shown how Symbolic Execution Trees can be used to perform static slicing. Symbolic Execution Trees can, therefore, be applied to any area that currently uses static slicing.

9.4.1 Dataflow Minimal Weiser Slices

Although a variety of Dependencies have been introduced in this thesis, all arguably as useful as a Weiser Slice, none is identical to a Dataflow Minimal Weiser Slice(see Section 3.13). The example in Section 3.13.1 (page 113)shows that occasionally *DTLD* gives rise to slices which do not contain statements which arguably should be included and are included in slices produced by Weiser's Algorithm. *DLD* on the other hand, because, unlike a Weiser slice, it is required to have exactly the same termination conditions as the original program, contains statements not included in Weiser slices. Further investigation is required to see whether the techniques introduced in this thesis can produce dataflow minimal slices that precisely satisfy the semantic relationship satisfied by slices produced by Weiser's algorithm.

9.4.2 Programs with Procedures

Although not yet implemented, the author believes that, like in the Parallel Algorithm [28], the Dataflow Dependence of programs with procedures can be computed by successive approximation. As a first approximation, each procedure call will be treated as *FAIL* . This will enable us to apply the *DTLD* algorithm to the body of each procedure p_i. The resulting symbolic execution tree of p_i, with necessary adjustments to handle parameter passing, can then be used in place of each call to p_i. This will produce a second approximation. This process is repeated until there is no further change in the *DTLD* of each procedure.

9.4.3 Dynamic Slicing

Dynamic Slicing [71, 5, 43] offers the potential for much thinner slices, since it is asking about dependencies pertaining to particular executions of a program. Since it captures all relevant executions, the Symbolic Execution Tree is ideal for computing such dynamic information. In dynamic slicing, the program is first executed to produce an *execution history.* An execution history is the sequence of nodes visited during this execution. All that we require to perform dynamic slicing is the path that was visited during a particular execution. The symbolic state at the leaf node of the symbolic execution tree corresponding to that path contains all the necessary dependence information pertaining to that particular execution sequence. If the execution sequence is longer than a path in the Symbolic Execution Tree, then, by theorem 7.4.1, we can shorten the execution sequence (where loops have been iterated more times than necessary) without loss of dependence information.

Part II

Appendices

Appendix A

Sample Outputs from the *DTLD* and the *DTVD* Algorithms

The schema comes first followed by the *DTLD* and then the *DTVD*. For example in the first one below, `("c", ["b1", "b2", "f3"])` means that variable `c` is dataflow terminating label dependent on each label in the set $\{b1, b2, f3\}$.

These examples can be found together with the implementation at:
`http://p69-122.unl.ac.uk/~seb/phd/`

```
begin
   while b1(i)
   do begin
    if b2(c)
    then begin
          c:=f3(y);
          z:=f4()
         end
    else ;
    i:=f5(i)
   end
end
```

```
("DTLD", ["DTLD"])
("c", ["b1", "b2", "f3"])
("i", ["b1", "f5"])
("z", ["b1", "b2", "f4"])
```

```
("DTVD", ["DTVD"])
("c", ["c", "i", "y"])
("i", ["i"])
("z", ["c", "i", "z"])
```

```
while b1(x,y)
do if b2(x)
   then x:=f(x,y)
   else y:=g(y)
```

```
("DTLD", ["DTLD"])
("x", ["b1", "b2", "f"])
("y", ["b1", "b2", "f", "g"])
```

```
("DTVD", ["DTVD"])
("x", ["x", "y"])
("y", ["x", "y"])
```

```
!john howroyd example!
while b1(q)
do     while b2(p)
        do begin
            if b3(x)
            then z:=f4()
            else p:=f5();
            if b6(y)
            then begin
                   x:=f7();
                   y:=f8()
                 end
            else q:=f9()
           end
```

```
 ("DTLD", ["DTLD"])
("p", ["b1", "f5"])
("q", ["b1", "f9"])
("x", ["b1", "f7"])
("y", ["b1", "f8"])
("z", ["b1", "f4"])

("DTVD", ["DTVD"])
("p", ["p", "q"])
("q", ["q"])
("x", ["q", "x"])
("y", ["q", "y"])
("z", ["q", "z"])
```

```
if f1(i)
then while f1(i)
     do
       begin x:=f2(x);
             i:=f6(i)
       end
else x:=f4()

("DTLD", ["DTLD"])
("i", ["f1", "f6"])
("x", ["f1", "f2", "f4", "f6"])

("DTVD", ["DTVD"])
("i", ["i"])
("x", ["i", "x"])

while f1()
do z:=f2()

("DTLD", ["DTLD"])
("z", nil)

("DTVD", ["DTVD"])
("z", ["z"])

i:=f1()

("DTLD", ["DTLD"])
```

```
("i", ["f1"])

("DTVD", ["DTVD"])
("i", nil)




while b1(j)
do begin
    i:=b2();
    while  b3(i)
    do begin
          z:=f4();
          i:=f5(i)
       end;
    j:=f6(j)
  end
!does the final value of z depend on 5?!
!i don't think so!

("DTLD", ["DTLD"])
("i", ["b1", "b2", "b3", "f5"])
("j", ["b1", "f6"])
("z", ["b1", "b2", "b3", "f4"])

("DTVD", ["DTVD"])
("i", ["i", "j"])
("j", ["j"])
("z", ["j", "z"])

while b1(j)
```

```
do begin
   while  b2(i) do z:=f3(z);
  j:=f4(j)
end

("DTLD", ["DTLD"])
("j", ["b1", "f4"])
("z", nil)

("DTVD", ["DTVD"])
("j", ["j"])
("z", ["z"])

begin
 while b1(j)
 do begin
     i:=f2();
     while  b3(i)
     do begin
          z:=f4();
          i:=f5(i)
        end;
      j:=f6(j)
    end;
 if b1(j)
 then z:=f7()
 else z:=f8()
end
!does the final value of z depend on f5?!
!i don't think so!
```

```
("DTLD", ["DTLD"])
("i", ["b1", "b3", "f2", "f5"])
("j", ["b1", "f6"])
("z", ["f8"])

("DTVD", ["DTVD"])
("i", ["i", "j"])
("j", ["j"])
("z", nil)



!this is a nice example
the algorithm is clever enough to know
that if the loop terminates then b1(j) must be false
so z does not depend on f4 but it does depend on f5.!

begin
 while b1(j)
 do begin
     z:=f2(z);
     j:=f3(j)
   end;
 if b1(j)
 then z:=f4()
 else z:=f5()
end

("DTLD", ["DTLD"])
("j", ["b1", "f3"])
("z", ["f5"])
```

```
("DTVD", ["DTVD"])
("j", ["j"])
("z", nil)
```

```
if b1(i)
then while b1(i)
     do x:=f2(x)
else x:=f4()
```

```
("DTLD", ["DTLD"])
("x", ["f4"])
```

```
("DTVD", ["DTVD"])
("x", nil)
```

```
while b1(i) do x:=f2(x)
```

```
("DTLD", ["DTLD"])
("x", nil)
```

```
("DTVD", ["DTVD"])
("x", ["x"])
```

```
while b1() do  x:=f2()
```

```
("DTLD", ["DTLD"])
("x", nil)
```

```
("DTVD", ["DTVD"])
("x", ["x"])




!John Howroyd's first example!
! A counter example to
  z DVD {x,y} => z DVD x or z DVD y

  In this example z DVD {x,y} and not (z DVD x) and not (z DVD y)

  (assuming that for z DVD K in p there must exist two terminating
   states s1 and s2 differing at most on variables in K,
   and a program q (data flow equivalent to p), such that the final
   value of z in s1 differs from that in s2.)

  Using the web-published DD slicer we should get that z slice includes
3 and 4,
  but as it doesn't depend on either 'separately' we hypothesize that it
will fail to include these'.
!

begin
  q :=f1();
  while b2(q)
    begin
    h:=f3(x);
    k:=f4(y);
    z:=f5();
    p:=f6();
    while b7(p)
```

```
    do  begin
         if b8(h)
         then z:=f9()
         else z:=f11() ;
         if b13(k)
         then
           begin
            h:=f14() ;
            k:=f15()
           end
         else
         q:=f17()
      end
    end
end

("DTLD", ["DTLD"])
("h", nil)
("k", ["f4"])
("p", ["f6"])
("q", ["f1"])
("z", ["f5"])

("DTVD", ["DTVD"])
("h", ["h"])
("k", ["y"])
("p", nil)
("q", nil)
("z", nil)
```

```
if b1(j)
then i:=f2()
else x:=f3()
```

```
("DTLD", ["DTLD"])
("i", ["b1", "f2"])
("x", ["b1", "f3"])
```

```
("DTVD", ["DTVD"])
("i", ["i", "j"])
("x", ["j", "x"])
```

```
   if b1(i)
   then
    if b5(d)
    then c:=f3(y)
    else
   else
```

```
("DTLD", ["DTLD"])
("c", ["b1", "b5", "f3"])
```

```
("DTVD", ["DTVD"])
("c", ["c", "d", "i", "y"])
```

```
    if b1(c)
    then x:=f4(y)
```

```
    else x:=f4(y)


("DTLD", ["DTLD"])
("x", ["f4"])

("DTVD", ["DTVD"])
("x", ["y"])


begin
if b1(p) then FAIL else;
if b1(p) then else FAIL;
x:=f2(z)
end

("DTLD", ["DTLD"])
("x", nil)

("DTVD", ["DTVD"])
("x", nil)


begin
  z:=f1(a,b);
  while b2(p) do;
end


("DTLD", ["DTLD"])
("z", ["f1"])
```

```
("DTVD", ["DTVD"])
("z", ["a", "b"])
```

```
begin
if b1(p)
then z:=f2(k)
else;
while b1(p) do ;
end
```

```
("DTLD", ["DTLD"])
("z", nil)
```

```
("DTVD", ["DTVD"])
("z", ["z"])
```

```
   while b1(i)
   do begin
       if b2(c)
       then c:=f3(y)
       else z:=f4();
       i:=f5(i)
     end
```

```
("DTLD", ["DTLD"])
("c", ["b1", "b2", "f3"])
```

```
("i", ["b1", "f5"])
("z", ["b1", "b2", "f3", "f4", "f5"])
```

```
("DTVD", ["DTVD"])
("c", ["c", "i", "y"])
("i", ["i"])
("z", ["c", "i", "y", "z"])
```

```
while b1(j)
do begin
    i:=f2();
    while  b3(i)
    do begin
        z:=f4(z);
        i:=f5(i)
       end;
   z:=f3(i,z);
   i:=f6(i);
   j:=f7(j)
  end
```

```
("DTLD", ["DTLD"])
("i", ["b1", "b3", "f2", "f5", "f6"])
("j", ["b1", "f7"])
("z", ["b1", "b3", "f2", "f3", "f4", "f5", "f7"])
```

```
("DTVD", ["DTVD"])
("i", ["i", "j"])
("j", ["j"])
("z", ["j", "z"])
```

```
while b1(j)
do begin
    i:=f2();
    z:=f3(i,z);
    i:=f4(i);
    j:=f5(j)
  end
```

```
("DTLD", ["DTLD"])
("i", ["b1", "f2", "f4"])
("j", ["b1", "f5"])
("z", ["b1", "f2", "f3", "f5"])
```

```
("DTVD", ["DTVD"])
("i", ["i", "j"])
("j", ["j"])
("z", ["j", "z"])
```

```
begin
 if b1(i)
 then x:=f2(y)
 else x:=f3(z);
 if b4(d)
 then c:=f5(x)
 else c:=f5(x)
end
```

```
("DTLD", ["DTLD"])
("c", ["b1", "f2", "f3", "f5"])
("x", ["b1", "f2", "f3"])
```

```
("DTVD", ["DTVD"])
("c", ["i", "y", "z"])
("x", ["i", "y", "z"])
```

Appendix B

Programs

B.1 Complete Hope Program for $DTVD$ and$DTLD$

```
#! /usr/local/bin/hope -f
!~/newphd/programs/newboth.hop
!Here we stop unfolding loops when only the data dependency does not
!change. We keep the most unfolded - if you see what I mean.
!Much faster but is it right!?
!I feel that if the datadependency has not changed then nor will the
!control dependency in the next unfolding

uses list,lists,set,moresetops,pfun,ctype,types,settolist;

type name == list(char);
data  delta ==  va name ++ complex (name # (set delta)) ++ botdelta;

singleton: alpha -> set alpha;
singleton x <= x & empty;
update: (alpha -> beta) -> alpha -> beta -> (alpha -> beta);
update f x y z <= if z=x
                  then y
                  else f z;
```

```
data statement ==
 FAIL ++
 ass(name X delta) ++
 ife(delta X (list statement) X (list statement)) ++
 while(delta X (list statement));


data token == sym char  ++ str (list char ) ++ setvar(set delta);
parsestatement: (list token) -> (list statement) # (list token);
parsestatementlist: (list token) ->(list statement) # (list token);
parsestatement
((str V)::((sym ':')::((sym '=')::((str L)::((setvar S)::l)))))
<=  ([ass(V,complex(L,S))],l);

parsestatement((str "FAIL")::l) <=([FAIL],l);

parsestatement
((str "if")::((str L)::((setvar S)::((str "then")::l))))
<=  let(a,e::b) == parsestatement(l) in
      let(c,d) == parsestatement(b) in
       ([ife(complex(L,S),a,c)],d);

parsestatement
((str "while")::((str L)::((setvar S)::((str do) :: l))))
<=  let(a,b) == parsestatement(l) in
       ([while(complex(L,S),a)],b);

parsestatement((str "begin")::l) <=
let (c,m) == parsestatementlist(l)
in if m = nil
```

```
   then error "end expected"
   else let e::d== m
        in if e=(str "end")
           then (c,d)
           else error "end expected";



parsestatement(x::l) <= (nil,x::l);


parsestatement(nil) <= (nil,nil);


parsestatementlist(x::l) <=  let (a,b::c) ==   parsestatement(x::l)
                                 in  if b = (sym ';')
                                     then (
                                          let (e,f) == parsestatementlist(c)
                                          in (a<>e,f)
                                           )
                                    else (a,b::c);



lexstring:list(char) -> list(char) # list(char);
lexstring(nil) <= (nil,nil);
lexstring(x::l) <=   if  isalnum x
                         then (let (c,d) == lexstring(l) in (x::c,d))
                         else (nil,x::l);


skipcomment:list(char) -> list(char);
skipcomment(nil) <= nil;
skipcomment(x::l) <= if x /= '!'
                       then skipcomment(l)
                       else l;
```

```
lexpass1:list(char) -> list(token);
lexpass1(nil) <= nil;
lexpass1(x::l) <= if x = ' ' or x = '\n'
                     then lexpass1(l)
                     else if isalnum x
                          then (
                               let (a,b) == lexstring(x::l)
                               in str(a):: lexpass1(b)
                               )
                          else if x='!'
                                    then lexpass1(skipcomment(l))
                                    else
if (x='(') or (x=')') or (x=',') or (x ='=') or (x=':') or (x=';')
then                                        (sym x):: lexpass1(l)
else error("illegal symbol in input");

makesetvar:list(token) -> (set delta) # list(token);



lexpass2:list(token) -> list(token);
lexpass2(nil) <= nil;
lexpass2(x::l) <= if x /= sym('(')
                     then x::lexpass2(l)
                     else let (a,e::b) == makesetvar(l)
                          in  setvar(a)::lexpass2(b);

makesetvar(nil) <= (empty,nil);

makesetvar(str(x)::sym(',')::l)<=  let (a,b) == makesetvar(l)
                                         in (((va x) & empty) U a,b);
```

```
makesetvar(str(x)::l) <= (((va x) & empty),l);

makesetvar(x::l) <= (empty,x::l);

parse: list(char) -> list(statement);
parse l <= let (a,b) == parsestatement(lexpass2(lexpass1(l))) in a;

lex: list(char) -> list(token);
lex l <= lexpass2(lexpass1(l));




data state== ok(name -> delta) ++ botstate;
data SET == leaf state   ++  node(SET X delta X SET);

type path == set delta X set delta;

evaldelta: state -> delta -> delta;
evaldelta botstate x <= botdelta;
evaldelta (ok sigma) botdelta <= botdelta;
evaldelta (ok sigma) (va x) <=   sigma x;
evaldelta (ok sigma) (complex (f,S)) <=
complex(f,mapset1(evaldelta (ok sigma) ,S));

updatestateinstate:state -> state -> state;
updatestateinstate (ok st1)  (ok st2)  <= ok((evaldelta (ok st1) o st2));
updatestateinstate x y <= botstate;

treeinstate:  SET ->state -> SET;
treeinstate (leaf sigma') sigma <= leaf (updatestateinstate sigma sigma');
treeinstate (node(t1,r,t2)) sigma  <=
```

```
node (treeinstate t1 sigma,evaldelta sigma r, treeinstate t2 sigma);

sequence:SET -> SET -> SET;
sequence (leaf sigma) t'  <=  treeinstate t' sigma;
sequence(node(t1,r,t2)) t' <=  node(sequence t1 t',r,sequence  t2 t');



prune: path -> SET -> SET;
prune (l,m) (leaf x) <= leaf x;
prune (l,m) (node(b1,r,b2)) <=
if (r isin l)
then prune (l,m) b1
else if (r isin m)
     then prune (l,m) b2
     else node(prune (r & l,m) b1, r, prune (l,r & m) b2);

simplify: SET -> SET;
simplify <= prune(empty,empty);

meaning:statement -> SET;
meaningl:list(statement) -> SET;

meaningl nil  <= leaf (ok va);
meaningl (x::l)  <= simplify (sequence (meaning x) (meaningl l));

meaning FAIL <= leaf botstate;

meaning (ass(x,e))   <=
leaf(ok (update va x (evaldelta (ok va) e)));

meaning (ife(e,l1,l2))  <=
```

```
simplify (node(meaningl l1, evaldelta  (ok va) e,meaningl l2));


dec addleft,addright: delta X (pfun path delta) -> (pfun path delta);
addleft(d,f) <= mapset(lambda ((a,b),c) => singleton((d & a,b),c),f);

addright(d,f) <= mapset(lambda ((a,b),c) => singleton((a,d & b),c),f);

applystate :name -> state -> delta;
applystate v (ok sigma) <= sigma v;
applystate v botstate <= botdelta;

pathfun: SET -> name  -> (pfun path delta);
pathfun (leaf  sigma) v <=  singleton((empty,empty),applystate v sigma);
pathfun (node (b1,r,b2)) v <=  addleft (r,pathfun b1 v) U addright (r,pathfun b2 v);



variables: delta ->  set name;
variablesset: (set delta) -> (set name);

variables (va x) <= x & empty;
variables (complex (f,S))   <=  variablesset (S);
variables botdelta <= empty;

variablesset S  <=mapset(variables,S);

labels: delta ->  set name;
labelsset: (set delta) -> (set name);

labels (va x) <= empty;
labels (complex (f,S))   <=  (singleton f) U labelsset (S);
labels botdelta <= empty;
```

```
labelsset S  <=mapset(labels,S);

names: delta ->  set name;
names(x) <= (labels x) U (variables x);

nameset: (set delta) -> (set name);
nameset S  <= mapset(names,S);



datadependent:  (pfun path delta) -> set name ;
datadependent  <= nameset o range;



differences: path X path -> set delta;
differences((p1,p1'),(p2,p2')) <= ((p1 intersect p2') U (p1' intersect p2));

allintersect: set delta  -> set name;
allintersect S <= if S = empty
                  then empty
                  else let (a,T)==choose S
                       in if (card S) = 1
                          then  (names a)
                          else (names a) intersect (allintersect T);


Tcontroldependent :  (pfun path delta) -> set name ;
Tcontroldependent f <=
mapset(lambda d1 => mapset(
```

```
lambda d2 =>
if
 (apply f d1) = (apply f d2) or (apply f d1)=botdelta  or (apply f d2)=botdelta
then empty
else allintersect (differences(d1,d2))
                                                  , domain f)
                              ,domain f) ;



DTslice:  (pfun path delta) -> (set name);
DTslice f <= (datadependent f) U  (Tcontroldependent f);



affected: statement -> set name;
affectedl: (list statement) -> set name;
affected bottom <= empty;
affected (ass(x,E)) <= singleton x;
affected (ife(E,s1,s2)) <= (affectedl s1) U (affectedl s2);
affected (while(E,s)) <= affectedl s;
affectedl nil <= empty;
affectedl (x::l) <= (affected x) U (affectedl l);


samedatadependency: (set name)->SET->SET->bool;
samedatadependency A t1 t2<=
makepfun A (datadependent o  (pathfun t1))=makepfun A (datadependent o (pathfun  t2));


samedependency: (set name)->SET->SET->bool;
```

```
samedependency A t1 t2<=
makepfun A (DTslice o  (pathfun t1))=makepfun A (DTslice o (pathfun  t2));

least: (set name) -> delta -> (list statement) -> statement -> SET;
least A b l s <=  let (t1,t2) == (meaning s, meaning (ife(b,l<>[s],[])))
                  in if samedatadependency A t1 t2
                     then t2
                     else least A b l (ife(b,l<>[s],[]));

!here we're stopping when datadpendency is same
!change to samedependency if I like

meaning (while(b,l))  <=
least (affected (while (b,l))) b l (ife(b,[FAIL],[]));



DTD: list statement -> pfun name (set name);
DTD l <=
makepfun (affectedl l)
(DTslice o (pathfun  (meaningl l)));



allvariables: statement -> set name;
allvariablesl: (list statement) -> set name;
allvariables bottom <= empty;
allvariables (ass(x,E)) <= x & (variables E);
allvariables (ife(E,s1,s2)) <= (variables E) U (allvariablesl s1) U (allvariablesl s2);
allvariables (while(E,s)) <= (variables E) U (allvariablesl s);
allvariablesl nil <= empty;
allvariablesl (x::l) <= (allvariables x) U (allvariablesl l);
```

```
alllabels: statement -> set name;
alllabelsl: (list statement) -> set name;
alllabels bottom <= empty;
alllabels (ass(x,E)) <=  (labels E);
alllabels (ife(E,s1,s2)) <= (labels E) U (alllabelsl s1) U (alllabelsl s2);
alllabels (while(E,s)) <= (labels E) U (alllabelsl s);
alllabelsl nil <= empty;
alllabelsl (x::l) <= (alllabels x) U (alllabelsl l);



sortofrangerestrict: set name ->  pfun name (set name) ->
pfun name (set name);
sortofrangerestrict S f <=
if f=empty
then empty
else  let ((a,b),g) == choose f
      in  (a,b intersect S) & (sortofrangerestrict S g);



DTLD: list statement -> pfun name (set name);
DTLD s <= sortofrangerestrict (alllabelsl s) (DTD s);


DTVD: list statement -> pfun name (set name);
DTVD s <= sortofrangerestrict (allvariablesl s) (DTD s);
k:list(char) -> list(name X set(name));
k(l) <= [("DTLD","DTLD"& empty)]<>
(settolist (DTLD(parse l))) <>[("DTVD","DTVD"&empty)] <> (settolist (DTVD(parse l)));

write(k input);
```

B.2 Auxiliary Hope Functions

```
infix isin :4;

infix minus :4;

infix intersect:4;

dec     isin     : alpha # set(alpha) -> truval;

--- x isin s <= if s = empty
                 then false
                 else let (y,l) == choose s
                      in   if x=y
                           then true
                           else x isin l;

dec intersect: (set alpha) X (set alpha) -> (set alpha);

s intersect t <= if t=empty
                 then empty
                 else let (a,b) == choose t
                      in (if (a isin s) then( a & empty) else empty) U (s intersect b);

dec minus: set (alpha) # set (alpha) -> set (alpha);

s minus t <= if s = empty
                  then empty
                  else let (a,v) == choose(s)
                        in if a isin t
                           then v minus t
                           else a & (v minus t);


dec mapset: (alpha -> set (beta)) # set(alpha) -> set(beta);

--- mapset(f,s) <= if s = empty
```

```
                   then empty
                   else let (x,y) == choose(s)
                        in f(x) U mapset(f,y);
dec mapset1: (alpha -> beta) # set(alpha) -> set(beta);
--- mapset1(f,s) <= if s=empty
                    then empty
                    else let (x,y) == choose(s)
                         in (f(x) & empty) U mapset1(f,y);

dec maplist: (alpha -> beta) # list(alpha) -> list(beta);
--- maplist(f,nil) <= nil;
--- maplist(f,x::l) <= f(x)::maplist(f,l);

dec maplisttoset: (alpha -> beta) # list(alpha) -> set(beta);
--- maplisttoset(f,nil) <= empty;
--- maplisttoset(f,x::l) <= (f(x) & empty) U maplisttoset(f,l);

type pfun alpha beta  == set(alpha X beta);
apply: pfun alpha beta -> alpha -> beta;
apply f z <= let ((a,b),g) == choose f
             in if a=z
                then b
                else apply g z;

makepfun: set alpha -> (alpha -> beta) -> pfun alpha beta;
makepfun S f <= if S= empty
                then empty
                else let (a,T) == choose S
                     in (a,f a) & (makepfun T f);
```

```
domain: pfun alpha beta -> set alpha;
domain f <= if f=empty
            then empty
            else  let ((a,b),g) == choose f
                  in a & domain g;

range: pfun alpha beta -> set beta;
range f <= if f=empty
           then empty
           else  let ((a,b),g) == choose f
                 in b & range g;




update:  alpha -> beta -> pfun alpha beta -> pfun alpha beta;
update x y f <=  if f=empty
                 then (x,y) & empty
                 else  let ((a,b),g) == choose f
                       in if x=a then (x,y) & g
                       else (a,b) & (update x y g);

restrict: set alpha ->  pfun alpha beta -> pfun alpha beta;
restrict S f <= if f=empty
                then empty
                else  let ((a,b),g) == choose f
                      in if a isin S
                         then (a,b) & (restrict S g)
                         else restrict S g;

rangerestrict: set beta ->  pfun alpha beta -> pfun alpha beta;
rangerestrict S f <= if f=empty
```

```
                then empty
                else  let ((a,b),g) == choose f
                      in if b isin S
                         then (a,b) & (rangerestrict S g)
                         else rangerestrict S g;

compose: (beta -> gamma) -> pfun alpha beta -> pfun alpha gamma;
compose f g  <= if g=empty
                then empty
                else  let ((a,b),h) == choose g
                      in  (a,f b) & (compose f h);

!the identity function with domain S.
idpfun:set alpha -> pfun alpha alpha;
idpfun S <=  makepfun S id;




override: pfun alpha beta -> pfun alpha beta -> pfun alpha beta;
override f g <=   (restrict ((domain f) minus (domain g)) f) U g;



dec settolist:set(alpha) -> list(alpha);
--- settolist(s) <= if s=empty
                    then []
                    else let (a,b) == choose(s)
                         in a::settolist(b);
```

Appendix C

Correctness of the Parallel Algorithm

C.1.1 Functional Networks

The processes networks used in the parallel slicing algorithm can be defined in terms of recursion equations, not over infinite streams as in [1], but over finite sets of variable names and node identifiers.

Example

Consider the process network described in subsection 2.3.4 with each arc and node labelled as follows:-

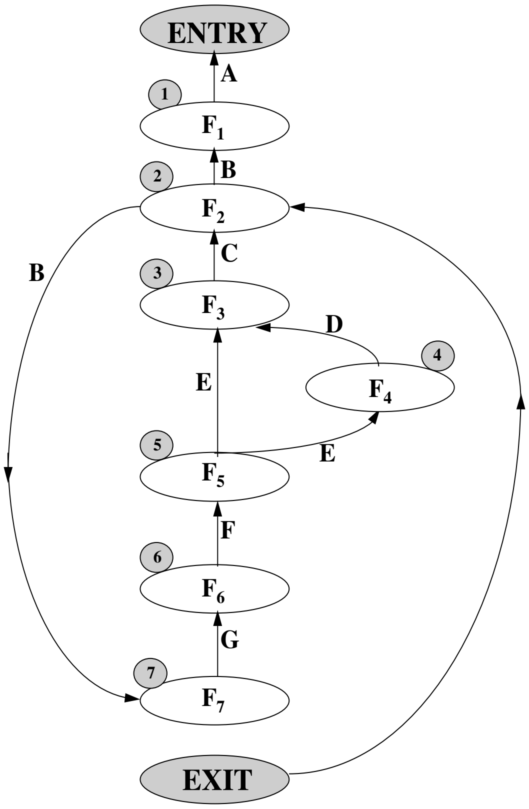


Figure C.1: The functional network derived from the example program

From the diagram, the following recursion equation is derived:-

$$G = F_7(F_2(F_3(F_5(F_6(G)) \cup F_4(F_5(F_6(G))))))$$

Where each F_i corresponds to the behaviour of the process i as a function on sets as described in subsection 2.3.1. i.e.

$F_i(S)$=

if S $\cap$ $(def(i) \cup \mathrm{C}(i)) \neq \emptyset$

then $(\mathrm{S} \backslash def(i)) \cup ref(i) \cup \{i\}$

else S

Inputs to processes are represented by arguments to the corresponding functions, F_i, and outputs of processes by the results of the corresponding functions. Clearly, different network topologies are achieved by composing the functions in different ways[1]. Notice that if a process has more than one input, then the argument of the corresponding function is the union of the individual inputs. In the above example there is one loop and hence only one equation. In general however, there will be an equation for each cycle in the Rcontrol flow graph.

Solving the Equations to Produce Slices

The equations, in isolation, represent the static properties of network. A solution to the equations represents a possible labelling with a set of variable names and node identifiers of all the arcs of the Rcontrol flow graph. For each arc, this label corresponds to the union of all messages that, in a valid implementation (see subsection C.1.1), will be transmitted along the communication channel represented by that arc.

Valid Implementations

In general, of course, there are many solutions to such equations. Following [78], a *valid implementation* of the parallel slicing algorithm is defined as one which produces the *least* solution to the equations; that is, the least solution relative to the partial order of arc–wise set inclusion, i.e., using $\sqsubseteq$ to denote the partial order, for networks L_1 and L_2, $L_1 \sqsubseteq L_2$ if and only if, for every arc, the corresponding label for L_1 is a subset of the corresponding label for L_2.

In order to produce the slice, the solution sought must be the least solution which contains the slice set as a subset of the label at the slice node (because the parallel slicing algorithm is initiated by the slice node outputting the slice set).

[1]The possibility of describing the network in this way suggests an implementation of the algorithm by compiling the Rcontrol flow graph into a functional program.

In subsection C.1.1 above, since the slice set is $\{\mathsf{c}\}$, the least solution to the recursion equation that has $\{\mathsf{c}\} \subseteq G$ is required.

C.1.2 Correctness of the Parallel Slicing Algorithm

In this subsection the parallel slicing algorithm is proved correct, in the sense that every statement included in a Weiser slice will also be included using the parallel slicing algorithm. Some preliminary results are first stated and proved.

Existence of Solutions

It is necessary to verify that solutions to such equations exist. This follows Kleene's first recursion theorem [78], from the fact that the functions F_i, introduced in subsection C.1.1 are monotonic with respect to set inclusion and hence can be solved by constructing Kleene chains.

Termination

It is also important to show that such recursion equation systems give rise to terminating computations. To do this, it must be shown that *finite* solutions always exist. This follows from the fact that the labelling of each arc must be contained in the finite set consisting of all node identifiers and variable names of the control flow graph.

It can easily be shown that the functions representing processes possess the *additive* property i.e.

$$F(A \cup B) = F(A) \cup F(B)$$

This ensures that, in a valid implementation of the parallel slicing algorithm, each process never need output the same value more than once, ensuring that it need not output messages indefinitely.

Correctness Proof

Definition C.1.1 ($\text{outputs}_{(V,i)}(n)$)

Let (V, i) be a slicing criterion for a control flow graph and let n be a node of the control flow graph,

$\text{Outputs}_{(V,i)}(n)$ is defined to be the union of all messages output by node n when slicing using the parallel slicing algorithm with respect to (V, i).
More rigorously, $\text{outputs}_{(V,i)}(n)$ is the labelling of all arcs emerging from n in the least solution of the equations derived from the control flow graph where the labelling of all arcs emerging from i contain V.

Lemma 1

Let (V, i) be a slicing criterion for a control flow graph then $V \subseteq \text{outputs}_{(V,i)}(i)$

Proof obvious.

Lemma 2

Let (V, i) be a slicing criterion for a control flow graph
If $K \subseteq \text{outputs}_{(V,i)}(b)$ then for all nodes, j, of the control flow graph, $\text{outputs}_{(K,b)}(j) \subseteq \text{outputs}_{(V,i)}(j)$

Proof

Since $K \subseteq \text{outputs}_{(V,i)}(b)$, the least solution containing V on all arcs emerging from node i contains K on all arcs emerging from node b.
For all nodes j, let $\text{outputs}_{(K,b)}(j) = X_j$
So by definition, the least solution containing K on all arcs emerging from b has X_j on all arcs emerging from j, for all nodes j.
So the least solution containing V on all arcs emerging from i has X_j on all arcs emerging from j, for all nodes j.
So for all nodes j of the control flow graph, $\text{outputs}_{(K,b)}(j) \subseteq \text{outputs}_{(V,i)}(j)$ as required.

The main theorem will now be proved. It is proved that every statement included in a Weiser slice will also be included in the slice obtained using the parallel slicing algorithm.

It shall be proved by induction that for all $n \geq 0$, for all slicing criteria C, for all nodes i that:-

1. $\mathrm{R}_C^n(i) \subseteq \mathrm{outputs}_C(i)$

   i.e. the label of the arc(s) from node i include the relevant variable for node i.

2. $j \in S_j^n \Rightarrow j \in \mathrm{outputs}_C(j)$

   i.e. if j is a relevant statement then it will output its node identifier.

Base Case

First it is proved that:-

3. $\forall i, c \quad \mathrm{R}_C^0(i) \subseteq \mathrm{outputs}_C(i)$

4. $\forall j, c \quad j \in S_C^0 \Rightarrow j \in \mathrm{outputs}_C(j)$

Proof of 3

Part 3 is proved by induction on the *maximum distance* of i from the slice node. The maximum distance from node i to node j is the maximum number of distinct nodes in a path from i to j.

First, if i is the slice node then by definition $\mathrm{R}_C^0(i) = V$ (V is the slice set) and by definition of the parallel slicing algorithm (subsections 2.3.2 and C.1.1), $V \subseteq \mathrm{outputs}_C(i)$.

If i is not the slice node, suppose for all arcs at a maximum distance $\leq N$ from the slice node that $\mathrm{R}_C^0(i) \subseteq \mathrm{outputs}_C(i)$.

Let i be a node at a maximum distance $N+1$ from the slice node, then by definition, all nodes which input to i are at a distance $\leq N$ from i and inductively, it can be concluded that:-

5. The inputs to i contains $\bigcup_{j \rightarrow_{RCFG} i} \mathrm{R}_C^0(j)$

Let $v \in \mathrm{R}_C^0(i)$, then, by definition,

either there exists a j such that $j \rightarrow_{RCFG} i$ and $v \notin def(i)$ and $v \in \mathrm{R}_c^0(j)$
in which case v is input to node i (by 5 above) and will be output by i
(by definition of process behaviour (subsections C.1.1 and 2.3.1)).

or $v \in ref(i)$ and there exists a j such that $def(i) \cap \mathrm{R}_C^0(j) \neq \emptyset$
in which case by 5 above and again by definition of process behaviour,
it follows that i outputs $ref(i)$ and hence i outputs v.

Concluding that $\mathrm{R}_C^0(i) \subseteq \mathrm{outputs}_C(i)$

Proof of 4

$$\text{By definition } S_C^0 = \{i \mid def(i) \cap \mathrm{R}_C^0(j) \neq \emptyset\}$$

$$\begin{aligned}
\text{hence } j \in S_C^0 \quad &\Rightarrow \quad \exists k \text{ such that } k \rightarrow_{RCFG} j \text{ and } def(j) \cap \mathrm{R}_C^0(k) \neq \emptyset \\
&\Rightarrow \quad \exists k \text{ such that } k \rightarrow_{RCFG} j \text{ and } def(j) \cap \mathrm{outputs}_C(k) \neq \emptyset \\
&\Rightarrow \quad j \in \mathrm{outputs}_C(j) \quad \text{by definition of process behaviour (C.1.1, 2.3.1)}
\end{aligned}$$

This concludes the proof of the base case.

Inductive Step

Now assume

6. $\forall i, C \;\; \mathrm{R}_C^N(i) \subseteq \mathrm{outputs}_C(i)$

and

7. $\forall j, C \;\; j \in S_C^N \Rightarrow j \in \text{outputs}_C(j)$

It must be proved that:-

8. $\forall i, C \;\; \mathrm{R}_C^{N+1}(i) \subseteq \text{outputs}_C(i)$ and

9. $\forall j, C \;\; j \in S_C^{N+1} \Rightarrow j \in \text{outputs}_C(j)$

Proof of 8

Weiser defines

$$\mathrm{R}_C^{K+1}(i) = \mathrm{R}_C^K(i) \cup \bigcup_{b \in \mathrm{B}_C^K} \mathrm{R}^0_{(b, ref(b))}(i)$$

and

$$S_C^{K+1} = \mathrm{B}_C^K \cup \{i \mid \exists j \text{ such that } i \rightarrow_{CFG} j \text{ and } def(i) \cap \mathrm{R}_C^{K+1}(j) \neq \emptyset\}$$

and

$$\mathrm{B}_C^K = \{b \mid \exists i \in S_C^K \text{ such that } b \text{ controls } i\}$$

Now let $v \in \mathrm{R}_C^{N+1}(i)$

$$\text{Assume } \exists b \in \mathrm{B}_C^N \text{ such that } v \in \mathrm{R}^0_{(b,ref(b))}(i)$$

Then by (3) it follows that

$$v \in \mathrm{outputs}_{(b,ref(b))}(i)$$

But, by definition, b controls a node in S_C^K. So by the induction hypothesis (7) this node will have output its node identifier. b will therefore receive this node identifier, and by definition of process behaviour (subsections C.1.1 and 2.3.1), b will output $ref(b)$.

$$\text{i.e. } ref(b) \subseteq \mathrm{outputs}_{(v,i)}(b)$$

$$\text{so } \mathrm{outputs}_{(b,ref(b))}(i) \subseteq \mathrm{outputs}_C(i) \text{ by Lemma 2 (subsection C.1.2)}$$

$$\text{so } v \in \mathrm{outputs}_C(i)$$

as required for proof of (8).

Proof of 9

Let $i \in S_C^{N+1}$,

then $i \in \mathrm{B}_C^N$ or $def(i) \cap \mathrm{R}_C^{N+1}(j) \neq \emptyset$ for some node j inputting to i.

if $\quad def(i) \cap \mathrm{R}_C^{N+1}(j) \neq \emptyset$

then $def(i) \cap \mathrm{outputs}_C(j) \neq \emptyset$ for some node j inputting to i which,

by definition of process behaviour (C.1.1 and 2.3.1), implies $i \in \mathrm{outputs}_C(i)$

or if $i \in \mathrm{B}_C^N$

then i controls an element j say of S_C^N

but by induction hypothesis (7), $j \in \mathrm{outputs}_C(i)$

This concludes the proof.

Appendix D

Proof of the $DTVD$ algorithm for Loop–free Schemas

,

Lemma D.0.1 *$y \in$ Vdatadepends s x implies there exist a path π and a program $p \in [s]$ and two states σ and σ' differing only at y such that such that*

$$(\textit{satisfy}\ s\ p\ \sigma\ \pi)$$
$$\textit{and}$$
$$(\textit{satisfy}\ s\ p\ \sigma'\ \pi)$$

and

$$\perp \neq \textit{evalsym}\ s\ p\ \sigma\ (\textit{pfun}\ s\ \pi\ x) \neq \textit{evalsym}\ s\ p\ \sigma'(\textit{pfun}\ s\ \pi\ x) \neq \perp.$$

Proof:

Since $y \in \textit{Vdatadepends}\ s\ x$, there exists a path π such that $y \in \textit{variables}(\textit{pfun}\ s\ \pi\ x)$ We can therefore choose two states σ and σ' differing only at y such that

$$\perp \neq \textit{evalsym}\ s\ p\ \sigma\ (\textit{pfun}\ s\ \pi\ x) \neq \textit{evalsym}\ s\ p\ \sigma'(\textit{pfun}\ s\ \pi\ x) \neq \perp.$$

By Assumption 3.4.1, we can find values for all the predicate functions occurring as outermost labels in π such that

$$(\textit{satisfy}\ s\ p\ \sigma\ \pi)$$

$$\text{and}$$
$$(satisfy\ s\ p\ \sigma'\ \pi)$$

as required.

Lemma D.0.2 *Let δ_i, $i \in \{1 \cdots n\}$ be a set of n non–predicate symbolic values obtained from s. Let*

$$V = \{v_i \mid i \ \in \ 1 \cdots n\}$$

be a set of n integers then there exists a program p in $[s]$ and a state σ and such that for all $i \in \{1 \cdots n\}$,

$$v_i = evalsym\ s\ p\ \sigma\ \delta_i$$

Proof:

Lemma D.0.3 *Let δ_i, $i \in \{1 \cdots n\}$ be a set of distinct non–predicate symbolic values obtained from s that such that for all $i \in \{1 \cdots n\}$, $y \in$ variables δ_i . Let*

$$V = \{v_i \mid i \ \in \ 1 \cdots n\}$$
$$and$$
$$V' = \{v'_i \mid i \ \in \ 1 \cdots n\}$$

be two sets of n distinct integers with

$$V \cap V' = \emptyset$$

then there exists a program p in $[s]$ and a two states σ and σ' differing only at y such that for all $i \in \{1 \cdots n\}$,

$$v_i = evalsym\ s\ p\ \sigma\ \delta_i$$
$$and$$
$$v'_i = evalsym\ s\ p\ \sigma'\ \delta_i.$$

Proof:

Induction on the maximum depth of the δ_i.

<u>Base Case</u>

The δ_i are all variables then $n = 1$. Simply pick any σ, σ such that

$$\sigma\ y = v_1$$
and
$$\sigma'\ y = v_1'\ .$$

Induction Hypothesis Assume true for all δ_i of depth $< \ m$. Let δ_i, $i \in \{1 \cdots k\}$ be a set of non–predicate symbolic values that have maximum depth of m and are such that

Suppose there are k proper sub–expressions , δ' of $\{\delta_i\}$ that mention y. Let

$$U = \{u_i \mid i \ \in \ 1 \cdots k\}$$
and
$$U' = \{u_i' \mid i \ \in \ 1 \cdots k\}$$

be two sets of k distinct integers with

$$U \cap U' = \emptyset$$

By induction hypothesis, we can choose σ and σ' differing only at y such that for all $i \in \{1 \cdots k\}$,

$$u_i = evalsym\ s\ p\ \sigma\ \delta_i'$$
and
$$u_i' = evalsym\ s\ p\ \sigma'\ \delta_i'.$$

Let $\delta_i^k = f_i(S)$ be the elements of $\{\delta_i\}$ whose depth is k.

$$evalsym\ s\ p\ \sigma\ f_i(S) = \mathcal{E}[\![p\ f_i]\!] \bigcup_{\delta \in S} varof(s, \delta) \mapsto (evalsym\ s\ p\ \sigma\ \delta)$$
and
$$evalsym\ s\ p\ \sigma'\ f_i(S) = \mathcal{E}[\![p\ f_i]\!] \bigcup_{\delta \in S} varof(s, \delta) \mapsto (evalsym\ s\ p\ \sigma'\ \delta)$$

Since $y \in variables\ f_i(S)$,

$$\bigcup_{\delta \in S} varof(s, \delta) \mapsto (evalsym\ s\ p\ \sigma\ \delta) \text{ and } \bigcup_{\delta \in S} varof(s, \delta) \mapsto (evalsym\ s\ p\ \sigma'\ \delta)$$

are two states that are not equal and have not occurred in the evaluation of any proper–subexpressions containing y.

By Assumption 3.4.1, the expressions in p corresponding to f_i can be chosen as required.

Lemma D.0.4 *Let π and π' be paths such that*

$$y \in \bigcap_{\delta \in diffs(\pi,\pi')} (variables\ \delta)$$

and

$$y \notin variables(pfun\ s\ \pi\ x)\ and\ y \notin variables(pfun\ s\ \pi'\ x)$$

and

$$\perp \neq pfun\ s\ \pi\ x \neq pfun\ s\ \pi'\ x \neq \perp$$

then there exists a program $p \in [s]$ and two states σ and σ' differing only at y such that such that

$$(satisfy\ s\ p\ \sigma\ \pi)$$

and

$$(satisfy\ s\ p\ \sigma'\ \pi')$$

and

$$\perp \neq evalsym\ s\ p\ \sigma\ (pfun\ s\ \pi\ x) \neq evalsym\ s\ p\ \sigma'(pfun\ s\ \pi'\ x) \neq \perp.$$

Proof:

By Lemma D.0.3(page 282), and Assumption 3.4.1, there exist two states differing only at y such that

$$(satisfy\ s\ p\ \sigma\ \pi)$$

and

$$(satisfy\ s\ p\ \sigma'\ \pi')$$

and since $\perp \neq pfun\ s\ \pi\ x \neq pfun\ s\ \pi'\ x \neq \perp$ we can choose these states such that

$$\perp \neq evalsym\ s\ p\ \sigma\ (pfun\ s\ \pi\ x) \neq evalsym\ s\ p\ \sigma'(pfun\ s\ \pi'\ x) \neq \perp.$$

D.0.3 Proof of *DTVD* Algorithm

Theorem D.0.1 *Given a loop–free schema s,*

$$y \in DTVD\ s\ x \iff y \ \in \ \text{DTVslice}\ \mathcal{S}[\![s]\!]\ x$$

Proof:

We must show that there exists a program p in $[s]$ and two states σ and σ' differing only at y in s and a state σ such that

$$\perp \neq \mathcal{M}[\![p]\!]\sigma\ x \neq \mathcal{M}[\![p]\!]\sigma'\ x \neq \perp$$

$$\Longleftrightarrow$$

$$y \ \in \ \ Vdatadepends\ \mathcal{S}[\![s]\!]\ x \cup VTcontrols\mathcal{S}[\![s]\!]\ x$$

$\Longrightarrow$

Assume that there exists a program p in $[s]$ and two states σ and σ' differing only at y in s and a state σ such that

$$\perp \neq \mathcal{M}[\![p]\!]\sigma\ x \neq \mathcal{M}[\![p]\!]\sigma'\ x \neq \perp$$

By Theorem 5.4.1(page 160), there exist unique paths π and π' in $dom\ (pfun\ s)$ such that

$$evalsym\ s\ p\ \sigma\ \ (pfun\ s\ \pi\ x) \neq evalsym\ s\ p\ \sigma'\ \ (pfun\ s\ \pi'\ x)$$

such that

$$satisfy\ s\ p\ \sigma\ \pi \text{ and } satisfy\ s\ p\sigma'\ \pi'.$$

<u>Case 1</u> if $\pi = \pi'$ then $y \in variables(pfun\ s\ \pi\ x)$ since σ and $sigma'$ differ only at y. (Otherwise $evalsym\ s\ p\ \sigma\ \ (pfun\ s\ \pi\ x)$ and $evalsym\ s\ p\ \sigma'\ \ (pfun\ s\ \pi\ x)$ would have to be identical). therefore $y \in Vdatadepends\ s\ x$ as required.

<u>Case 2</u> if $\pi \neq \pi'$

Again if $(pfun\ s\ \pi\ x) = (pfun\ s\ \pi'\ x)$ then either $y \in variables(pfun\ s\ \pi\ x)$ and $y \in variables(pfun\ s\ \pi'\ x)$ so $y \in Vdatadepends\ s\ x$ as before.

Assume $(pfun\ s\ \pi\ x) \neq (pfun\ s\ \pi'\ x)$ and $y \notin variables((pfun\ s\ \pi\ x))$ and $y \notin variables((pfun\ s\ \pi'\ x))$. For all $\delta \in diffs(\pi, \pi')$, $y \in variables\ \delta$ and therefore y is in $VTcontrols\ s\ x$ as required.

$\Longleftarrow$

Assume $y \in Vdatadepends\ \mathcal{S}[\![s]\!]\ x \cup VTcontrols\mathcal{S}[\![s]\!]\ x$.

<u>Case 1</u> $\quad y \in Vdatadepends\ s\ x$

so there must exist a path $\pi = (\pi_t, \pi_f)$ with $(pfun\ s\ \pi\ x)$ such that $y \in variables(pfun\ s\ \pi\ x)$. by Lemma D.0.1(page 281), there exist two states σ and σ' differing only at y such that and a program $p \in [s]$ such that

$$(satisfy\ s\ p\ \sigma\ \pi)$$

and

$$(satisfy\ s\ p\ \sigma'\ \pi)$$

and

$$evalsym\ s\ p\ \sigma\ (pfun\ s\ \pi\ x) \neq evalsym\ s\ p\ \sigma'(pfun\ s\ \pi\ x).$$

as required.

Case 2 $\quad y \in VTcontrols\ s\ x$

so there exist two paths π and π' with $y \in variables \delta$ for all $\delta \in diffs(\pi, \pi') \neq \emptyset$ such that

$$pfun\ s\ \pi\ x \neq pfun\ s\ \pi'\ x$$

We can assume that neither $y \notin variables(pfun\ s\ \pi\ x)$ and $y \notin variables(pfun\ s\ \pi'\ x)$ since otherwise $y \in (Vdatadepends\ s\ x)$ which we have already considered. The result then follows immediately from Lemma D.0.4(page 284).

Bibliography

[1] Abramsky, S. Reasoning about concurrent systems. In *Distributed Computing* (London, 1984), pp. 307–319.

[2] Agrawal, H. On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Orlando, Florida, June 20–24 1994), pp. 302–312. Proceedings in SIGPLAN Notices, 29(6), June 1994.

[3] Agrawal, H., DeMillo, R. A., Pan, H., Spafford, E. H., and Viravan, C. Spyder project. URL `http://www.cs.purdue.edu/homes/spaf/spyder.html`.

[4] Agrawal, H., DeMillo, R. A., and Spafford, E. H. Debugging with dynamic slicing and backtracking. *Software – Practice and Experience 23*, 6 (June 1993), 589–616.

[5] Agrawal, H., and Horgan, J. R. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, June 1990), pp. 246–256.

[6] Bailey, R. *Functional programming with Hope.* Ellis Horwood, 1990.

[7] Ball, T., and Horwitz, S. Slicing programs with arbitrary control–flow. In 1^{st} *Conference on Automated Algorithmic Debugging* (Linköping, Sweden, 1993), P. Fritzson, Ed., Springer, pp. 206–222. Also available as Uniersity of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992.

[8] BECK, J., AND EICHMANN, D. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)* (1993), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 509–518.

[9] BEIZER, B. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[10] BELL, G., AND MUNRO, M. Using dynamic analysis to improve static analysis. In 2^{nd} *UK Workshop on Program Comprehension* (Durham, UK, July 1996), M. Munro, Ed.

[11] BENNETT, K., AND MORTIMER, R. Maintenance and abstraction of program data using formal transformations. In *IEEE International Conference on Software Maintenance* (1996), IEEE Computer Society Press, Los Alamitos, California, USA.

[12] BIEMAN, J. M., AND OTT, L. M. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20*, 8 (Aug. 1994), 644–657.

[13] BINKLEY, D. W. The application of program slicing to regression testing. In *Journal of Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998. to appear.

[14] BINKLEY, D. W. Computing amorphous program slices using dependence graphs and a data-flow model. In *ACM Symposium on Applied Computing* (The Menger, San Antonio, Texas, U.S.A., 1999), ACM Press, New York, NY, USA. to appear.

[15] BINKLEY, D. W., AND GALLAGHER, K. B. Program slicing. In *Advances of Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1996, pp. 1–50.

[16] BIRKHOFF, G., AND MCLANE, S. *A Survey of Modern Algebra.* Macmillan, 1949.

[17] BJØRNER, D., ERSHOV, A. P., AND JONES, N. D. *Partial evaluation and mixed computation.* North–Holland, 1987.

[18] BULL, T. A transformation system for maintenance — turning theory into practice. In *IEEE International Conference on Software Maintenance (ICSM'92)* (1992), IEEE Computer Society Press, Los Alamitos, California, USA.

[19] Canfora, G., Cimitile, A., and De Lucia, A. Conditioned program slicing. In *Journal of Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998. to appear.

[20] Canfora, G., Cimitile, A., De Lucia, A., and Lucca, G. A. D. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)* (Victoria, Canada, Sept. 1994), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 424–433.

[21] Cartwright, R., and Felleisen, M. The semantics of program dependence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1989), pp. 13–27.

[22] Cheng, J. Slicing concurrent programs – a graph–theoretical approach. In 1^{st} *Automatic Algorithmic Debugging Conference (AADEGUB'93)* (1993), P. Fritzson, Ed., pp. 223–240. Appears as Springer Lecture Notes in Computer Science vol 749.

[23] Choi, J., and Ferrante, J. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems 16*, 4 (July 1994), 1097–1113.

[24] Church, A. The calculi of lambda–conversion. *Annals of Mathematical Studies 6* (1951).

[25] Cimitile, A., De Lucia, A., and Munro, M. Qualifying reusable functions using symbolic execution. In *Proceedings of the* 2^{nd} *working conference on reverse engineering* (Toronto, Canada, 1995), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 178–187.

[26] Cimitile, A., De Lucia, A., and Munro, M. A specification driven slicing process for identifying reusable functions. *Software maintenance: Research and Practice 8* (1996), 145–178.

[27] Coen-Porisini, A., De Paoli, F., Ghezzi, C., and Mandrioli, D. Software specialization via symbolic execution. *IEEE Transactions on Software Engineering 17*, 9 (Sept. 1991), 884–899.

[28] Danicic, S., Harman, M., and Sivagurunathan, Y. A parallel algorithm for static program slicing. *Information Processing Letters 56*, 6 (Dec. 1995), 307–313.

[29] Dannenberg, and Ernst. Formal program verification using symbolic execution. *IEEE Transactions on Software Engineering 8* (Jan. 1982), 43–52.

[30] Day, N. A. *A Framework for Multi-Notation, Model-Oriented Requirements Analysis.* PhD thesis, Department of Computer Science, University of British Columbia, 1998.

[31] De Lucia, A., Fasolino, A. R., and Munro, M. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension* (Berlin, Germany, Mar. 1996), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 9–18.

[32] Dijkstra, E. W., and Dahl, O. *A discipline of programming.* Prentice Hall, 1972.

[33] Ershov, A. P. *On the essence of computation.* North–Holland Publishing, 1978, pp. 392–420.

[34] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*, 3 (July 1987), 319–349.

[35] Field, J., Ramalingam, G., and Tip, F. Parametric program slicing. In 22^{nd} *ACM Symposium on Principles of Programming Languages* (San Francisco, CA, 1995), pp. 379–392.

[36] Field, J., and Tip, F. Dynamic dependence in term rewriting systems and its application to program slicing. In *Journal of Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998. to appear.

[37] Futamura, Y. Partial evaluation of computation process – an approach to a compiler compiler. *Systems, Computers, Controls 2*, 5 (Aug. 1971), 721–728.

[38] Gallagher, K. B. *Using Program Slicing in Software Maintenance.* PhD thesis, University of Maryland, Baltimore, Maryland, December 1989.

[39] Gallagher, K. B. *Using program slicing in software maintenance.* PhD thesis, University of Maryland, College Park, Maryland, Jan. 1990.

[40] Gallagher, K. B. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the International Conference on Software Maintenance 1992* (Nov. 1992), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 236–244.

[41] Gallagher, K. B., and Lyle, J. R. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (Aug. 1991), 751–761.

[42] Glaser, H., Hankin, C., and Till, D. *Principles of Functional Programming.* Prentice Hall International, 1984.

[43] Gopal, R. Dynamic program slicing based on dependence graphs. In *IEEE Conference on Software Maintenance* (1991), pp. 191–200.

[44] Greibach, S. A. *Theory of Program Structures: Schemes, Semantics, Verification.* Springer-Verlag, Lecture Notes in Computer Science 36, 1975.

[45] Gupta, R., Harrold, M. J., and Soffa, M. L. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance* (Orlando, Florida, USA, 1992), IEEE Computer Society Press, pp. 299–308.

[46] Harman, M., and Danicic, S. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability 5*, 3 (Sept. 1995), 143–162.

[47] Harman, M., and Danicic, S. Amorphous program slicing. In 5^{th} *IEEE Internation Workshop on Program Comprehesion (IWPC'97)* (Dearborn, Michigan, USA, May 1997), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 70–79.

[48] Harman, M., and Danicic, S. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance 10*, 6 (1998), 415–441.

[49] Harman, M., Danicic, S., Sivagurunathan, B., Jones, B., and Sivagurunathan, Y. Cohesion metrics. In 8^{th} *International Quality Week* (San Francisco, May 29th – June 2nd. 1995), pp. Paper 3–T–2, pp 1–14.

[50] Harman, M., Danicic, S., and Sivagurunathan, Y. Program comprehension assisted by slicing and transformation. In 1^{st} *UK workshop on program comprehension* (Durham University, UK, July 1995), M. Munro, Ed.

[51] Harman, M., Danicic, S., Sivagurunathan, Y., and Simpson, D. The next 700 slicing criteria. In 2^{nd} *UK workshop on program comprehension* (Durham University, UK, July 1996), M. Munro, Ed.

[52] Hausler, P. A. Denotational program slicing. In 22^{nd}*, Annual Hawaii International Conference on System Sciences, Volume II* (Jan. 1989), pp. 486–495.

[53] Hecht, M. S. *Flow Analysis of Computer Programs.* Elsevier, 1977.

[54] Hoare, C. A. R. An axiomatic basis for computer programming. In *Communications of the ACM* (Oct. 1969), vol. 12.

[55] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice–Hall, 1985.

[56] Horwitz, S., Prins, J., and Reps, T. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Programming Languages* (Jan. 1988).

[57] Horwitz, S., Prins, J., and Reps, T. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems 11*, 3 (July 1989), 345–387.

[58] Horwitz, S., and Reps, T. Wisonsin program slicing project. URL `http://www.cs.wisc.edu/wpis/html/`.

[59] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988), pp. 25–46. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[60] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–61.

[61] Howden, W. E. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Trans. Softw. Eng. SE-3*, 4 (July 1977), 266–278.

[62] Huang, J. C. Instrumenting Programs for Symbolic-trace Generation. *Computer 13*, 12 (Dec. 1980), 17–23.

[63] Hudak, P., Jones, S. P., and Wadler, P. Report on the programming language haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices 27* (May 1992).

[64] Kahn, G. The semantics of a simple language for parallel programming. In *IFIP Congress 74* (1974), North–Holland, Amsterdam.

[65] Kahn, G., and MacQueen, D. B. Coroutines and networks of parallel processes. In *IFIP Congress 77* (1977), North–Holland, Amsterdam, pp. 993–998.

[66] Kamkar, M. *Interprocedural dynamic slicing with applications to debugging and testing.* PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993. Available as Linköping Studies in Science and Technology, Dissertations, Number 297.

[67] Kamkar, M. Application of program slicing in algorithmic debugging. In *Journal of Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998. to appear.

[68] Kamkar, M., Shahmehri, N., and Fritzson, P. Interprocedural dynamic slicing. In *Proceedings of the 4^{th} Conference on Programming Language Implementation and Logic Programming* (1992), pp. 370–384.

[69] King, J. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[70] Korel, B. Computation of dynamic slices for programs with arbitrary control flow. In 2^{nd} *International Workshop on Automated Algorithmic Debugging (AADEBUG'95)* (Saint–Malo, France, May 1995), M. Ducassé, Ed.

[71] Korel, B., and Laski, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (Oct. 1988), 155–163.

[72] Korel, B., and Rilling, J. Dynamic program slicing methods. In *Journal of Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998. to appear.

[73] Krinke, J., and Snelting, G. Validation of measurement software as an application of slicing and constraint solving. In *Journal of Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998. to appear.

[74] Lakhotia, A. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)* (1993), pp. 34–44.

[75] Liu, L., and Ellis, R. An approach to eliminating COMMON blocks and deriving ADTs from Fortran programs. Technical report, University of Westminster, UK, Feb. 1993.

[76] Longworth, H. D. Slice–based program metrics. Master's thesis, Michigan Technological University, 1985.

[77] Lyle, J. R., and Weiser, M. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications* (Peking, 1987), Institute of Electric and Electronic Engineers, pp. 877–882.

[78] Manna, Z. *Mathematical Theory of Computation.* McGraw–Hill, 1974.

[79] Ott, L. M. Using slice profiles and metrics during software maintenance. In *Proceedings of the* 10^{th} *Annual Software Reliability Symposium* (1992), pp. 16–23.

[80] Ott, L. M., and Bieman, J. M. Program slices as an abstraction for cohesion measurement, 1998. to appear.

[81] Ott, L. M., and Thuss, J. J. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium* (Baltimore, Maryland, USA, May 1993), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 71–81.

[82] Ottenstein, K. J., and Ottenstein, L. M. The program dependence graph in software development environments. *SIGPLAN Notices 19*, 5 (1984), 177–184.

[83] Reps, T., and Yang, W. The semantics of program slicing. Tech. Rep. Technical Report 777, University of Wisconsin, 1988.

[84] Shahmehri, N. *Generalized algorithmic debugging.* PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1991. Available as Linköping Studies in Science and Technology, Dissertations, Number 260.

[85] Simpson, D., Valentine, S. H., Mitchell, R., Liu, L., and Ellis, R. Recoup – Maintaining Fortran. *ACM Fortran forum 12*, 3 (Sept. 1993), 26–32.

[86] Sivagurunathan, Y., Harman, M., and Danicic, S. Slicing, I/O and the implicit state. In *3rd International Workshop on Automated Debugging (AADEBUG'97)* (Linköping, Sweden, May 1997), M. Kamkar, Ed., pp. 59–65.

[87] Snelting, G. Combining slicing and constraint solving for validation of measurement software. In *Static Analysis Symposium (SAS'96), LNCS 1145* (1996), pp. 332–348.

[88] Stoy, J. E. *Denotational semantics: The Scott–Strachey approach to programming language theory.* MIT Press, 1985. Third edition.

[89] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (Sept. 1995), 121–189.

[90] van der Waerden, B. L. *Moderne Algebra.* Ungar Publishing, 1943.

[91] Venkatesh, G. A. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Canada, June 1991), pp. 26–28. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[92] Weiser, M. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[93] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

[94] Weiser, M., and Lyle, J. R. *Experiments on slicing–based debugging aids.* Empirical studies of programmers, Soloway and Iyengar (eds.). Molex, 1985, ch. 12, pp. 187–197.

[95] Wilde, N., and Huitt, R. Maintenance support for object–oriented programs. *IEEE Transactions on Software Engineering 18*, 12 (1992), 1038–1044.

[96] Woodward, M. R., and Allen, S. P. Slicing algebraic specifications. *Information and Software technology 40*, 2 (1998), 105–118.

[97] Zhao, J., Cheng, J., and Ushijima, K. Static slicing of concurrent object-oriented programs. In 20^{th} *IEEE Annual International Computer Software and Applications Conference (COMPSAC'96)* (Seoul, Korea, August 1996), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 312–320.