Formal Aspects of Computing (1998) 3: 1–000

Slicing Programs in the Presence of Errors

Mark Harman[1], Dan Simpson[2] and Sebastian Danicic[1]

[1]School of Computing, University of North London, Holloway Road, London N7 8DB, UK;
[2]Department of Computing, University of Brighton, Moulsecoomb, Brighton, BN2 4GJ, UK.

Keywords: Program Slicing; Semantics; Errors

Abstract. Program slicing is a technique by which statements are deleted from a program in such a way as to preserve a projection of the original program's semantics. It is shown that slicing algorithms based upon traditional defined and referenced variable sets do not preserve a projection of strict semantics with respect to computations which cause errors. Rather, these approaches preserve a projection of the program's semantics which is lazy with respect to errors.
A modified version of defined and referenced variable sets is introduced, which provides the freedom to choose the form of semantics to be preserved.

1. Introduction

A slice is constructed with respect to a slicing criterion (V, n), where V is a set of variable identifiers and n is a point of interest within the program[1]. Definition 1 provides an informal definition of a program slice.

Definition 1. (Slice) Given a program p and a slicing criterion (V, n), a program slice p' is created by deleting statements from p. Any statement which does not affect the value of any variable in V when the next statement to be executed is at line number n may be deleted.

Correspondence and offprint requests to: Mark Harman, **Project** *Project* , School of Computing, University of North London, Holloway Road, London N7 8DB, UK. `m.harman@unl.ac.uk`

[1] In describing a slicing criterion it is conventional to label program points with line numbers. These line numbers are unique identifiers; one for each node in the program's Control Flow Graph [4].

Definition 1 makes the rather weak requirement that lines '*may* be deleted' rather than the stronger '*must* be deleted' because statement–minimal program slices are not, in general, computable [19].

This paper concerns the problem of constructing end-slices [11], where the line number co–ordinate of the slicing criterion is the 'end of the program'. The slicing criteria used in this paper are thus merely sets of variables. However, the results presented may be generalised to the more general case in which the line number may be any node of interest in the program's CFG.

The original definition of program slice given by Weiser [18], and its subsequent modifications by Korel and Laski [10] (for the dynamic paradigm) and by Venkatesh [17] and Tip [15] (for the quasi static paradigm) share the property that a slice is to be constructed by statement deletion, and that the slice should preserve a projection of the original program's semantics.

The motivation for slicing derives from the fact that the slice of a program is (usually) simpler, whilst it maintains the effect of the original upon the slicing criterion. This 'meaning projection' gives rise to many applications including cohesion measurement [2, 11], algorithmic debugging [9], re–engineering [12, 14], maintenance and debugging [5, 13] and program integration [8]. Tip [16] presents a detailed survey of the program slicing literature.

All slicing algorithms rely upon the computation of the sets of defined and referenced variables [1] for statements. Figure 8 sets out the algorithm by which these sets are calculated. Figures 1, 2 and 3 contain three example programs and slices constructed from them.

An interesting problem occurs when a slice is constructed for a program which may cause an error. Consider the example given in figure 2. Line one is removed in forming the slice on {y}, but *it should not be*. The problem is that line one causes an error, and so line two will not be executed. The value stored in y at the end of the program will not be 2, since the assignment to y never will be executed. The slice will not therefore preserve the effect of the original program upon the slicing criterion, and thus will not satisfy definition 1.

2. A Semantic Definition of Slicing

Using the semantics of a programming language it is possible to define 'end slicing' more formally. First the projection, $\mu \rhd V$, of the state to state mapping μ, onto the set of variables V, is defined as follows:

Definition 2. (Projection)

$$\forall \sigma, i. \quad (\mu \rhd V)\sigma i = \begin{cases} \mu\sigma i & \text{if } i \in V \\ \sigma i & \text{otherwise.} \end{cases}$$

Using projection, the equivalence which exists between a program and its end slice can be defined:

Definition 3. (Equivalence)
Let $\mathcal{C}$ be the meaning function of programs p and p'. p' is V *equivalent* to p if and only if $\mathcal{C}[\![p]\!] \rhd V = \mathcal{C}[\![p']\!] \rhd V$.

Two programs are V equivalent if they have an identical effect upon all variables in V.

Definition 4. (End Slice)

Given two programs p and p', p' is a slice of p with respect to slicing criterion V iff p is V equivalent to p' and p' can be obtained from p by the deletion of zero or more statements.

3. A Simple `while` Loop Language

Figures 4 and 5 describe the syntax and strict semantics of a simple `while` loop language, called **L** . N, I and E are the domains of numerals, identifiers and expressions respectively. The operator $|$ in figure 5 denotes integer division and $\mathcal{N}$ is a function from numerals to the natural numbers they denote.

L contains side–effect free expressions, which denote mappings from states to natural numbers, with the existence of division admitting the possibility that the meaning of an expression in a state is the error value, ϵ. The set of denotable values, V, is extended to a flat lattice in the usual way. Thus $V = \mathbb{N} \cup \{\epsilon, \perp\}$ with $\perp \sqsubseteq x$ for all x in V. States (in S) are mappings from identifiers to the values they denote.

Cartwright and Felleisen [3] show that Program Dependence Graphs (PDGs) exhibit a lazy semantics (in the usual sense; that is, with respect to termination). Specifically, the semantics of PDGs dominate the semantics of the sequential programs from which they are constructed. This result also applies to program slices, which dominate the semantics of the original programs from which they are constructed, with the result that slicing may remove non–termination.

Following Cartwright and Felleisen, Figure 6 introduces a semantics for **L** which is lazy with respect to errors. The definitions of $\mathcal{E}$, $\mathcal{O}$, V and S are identical to those in figure 5, as is the type of $\mathcal{C}$.

If an error (division by zero) occurs when the program p is executed in a state σ, then the strict denotation of p applied to σ is the 'everywhere error' state, $\lambda x.\epsilon$, whereas the lazy denotation is a state which yields ϵ only for those identifiers which are directly (or indirectly) affected by the error.

4. Slicing is Lazy with Respect to Errors

Consider the program and slice contained in figure 2. According to the strict definition of $\mathcal{C}$ (figure 5) the program and its 'slice' are not $\{\mathtt{y}\}$ equivalent, whereas according to the lazy definition of $\mathcal{C}$ (figure 6) the two programs are $\{\mathtt{y}\}$ equivalent.

Figure 7 shows the meaning of each example program and their corresponding slices under the lazy and strict semantic descriptions. Traditional slicing preserves a semantics which is lazy with respect to errors: errors are only preserved in the slice if they are *needed* by the computation of variables in the slicing criterion. A distinction may therefore be made between error–strict slicing and error–lazy slicing, with conventional algorithms producing error–lazy slices.

5. Constructing Error–Strict Slices

A statement s is included in an end-slice constructed with respect to V iff there is a transitive data or control dependence between s and the identifiers in V or any other statement included in the slice. Error producing statements and predicates will be included in a slice if they happen to define a variable upon which V (transitively) depends.

An obvious way to force the inclusion of error producing statements, therefore, is to create such a relationship. This can be achieved by introducing a pseudo variable *Err*, which is both defined and referenced by expressions involving division. Figure 9 gives the modified version of defined and referenced variables. A similar result can be achieved by introducing assignment statements [6], but this involves program modification prior to slice computation.

If the variable *Err* is included in the slicing criterion, then all those statements which could cause an error will be included in the slice produced (making it error–strict), otherwise the slice produced will be the conventional (error–lazy) slice.

6. Error Slices

Greater precision can be achieved by categorising the kinds of error which may occur, for example, null–pointer assignment, array index out of range, read past end of file and so on. A distinct pseudo variable can be used to denote each form of error. The slice constructed for each pseudo variable in isolation will therefore be an 'error–slice' containing all the statements which may give rise to the error in question.

The sums, averages and ratios of different forms of error could be used to provide a wealth of metrics calculable using approaches to slice–based cohesion measurement [2, 7]. It is also easy to define error–prone and error–free statements in terms of program slicing.

Definition 5. (Potential Error Statements)
Let $\{\epsilon_{E_1}, \ldots, \epsilon_{E_n}\}$ be a set of error variables which capture the error conditions $\{E_1, \ldots, E_n\}$ and let $\{S_1, \ldots, S_n\}$ be the set of slices of program p constructed for these error conditions. The set of *potential error statements* of program p with respect to $\{E_1, \ldots, E_n\}$ is $\bigcup_{i=1}^{n} S_i$.

Definition 6. (Error Prone Statements)
Let $\{\epsilon_{E_1}, \ldots, \epsilon_{E_n}\}$ be a set of error variables which capture the error conditions $\{E_1, \ldots, E_n\}$ and let $\{S_1, \ldots, S_n\}$ be the set of slices of program p constructed for these error conditions. The set of *error prone statements* of program p with respect to $\{E_1, \ldots, E_n\}$ is $\bigcap_{i=1}^{n} S_i$.

Definition 7. (Error Free Statements)
The complement of the subject program and its error prone statements with respect to a set of error conditions E, is the set of error free statements with respect to E.

These definitions could be used by a programmer during the testing phase of the software development life–cycle to identify those statements which require particularly rigorous testing.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, techniques and tools.* Addison Wesley, 1986.

[2] James M. Bieman and Linda M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

[3] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–27, 1989.

[4] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[5] Keith B. Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[6] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability*, 5:143–162, September 1995.

[7] Mark Harman, Sebastian Danicic, Balasubramaniam Sivagurunathan, Barry Jones, and Yogasundary Sivagurunathan. Cohesion metrics. In 8^{th} *International Quality Week*, pages Paper 3–T–2, pp 1–14, San Francisco, May 29th – June 2nd. 1995.

[8] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[9] Mariam Kamkar. *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.

[10] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[11] Arun Lakhotia. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)*, pages 34–44, 1993.

[12] Lulu Liu and Rod Ellis. An approach to eliminating COMMON blocks and deriving ADTs from Fortran programs. Technical report, University of Westminster, UK, February 1993.

[13] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications*, pages 877–882, Peking, 1987.

[14] Dan Simpson, Samuel H. Valentine, Richard Mitchell, Lulu Liu, and Rod Ellis. Recoup – Maintaining Fortran. *ACM Fortran forum*, 12(3):26–32, September 1993.

[15] Frank Tip. *Generation of Program Analysis Tools.* PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[16] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[17] Guda A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[18] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[19] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

1	`x := 1;`	1	`x := 1;`
2	`y := 2;`	2	
3	`z := x+1;`	3	`z := x+1`
4	`y := y+z`	4	
The Original		Slice Constructed for {`z`}	

Fig. 1. An Error–Free Program and One of its Slices

1	`x := 1 DIV 0;`	1	
2	`y := 2`	2	`y := 2`
The Original		Slice Constructed for {`y`}	

Fig. 2. An 'Incorrect' (i.e. error–lazy) Slice

1	`x := 1 DIV 0 ;`	1	`x := 1 DIV 0 ;`
2	`z := z+1 ;`	2	
3	`y := x+3`	3	`y := x+3`
The Original		Slicing Constructed for {`y`}	

Fig. 3. A Program with Errors that Produces a 'Correct' (i.e. error–strict) Slice

$$
\begin{array}{rcl}
E & ::= & E_1 \ \mathtt{BOp} \ E_2 \mid I \mid \mathrm{N} \\
\mathtt{BOp} & ::= & + \mid - \mid * \mid \mathrm{DIV} \\
C & ::= & I := E; \mid C_1;\, C_2 \mid \\
 & & \mathtt{if}\, E\, \mathtt{then}\, C\, \mathtt{fi} \mid \mathtt{while}\, E\, \mathtt{do}\, C\, \mathtt{od}
\end{array}
$$

Fig. 4. The Syntax of **L**

$$
\begin{array}{rcl}
\mathcal{O} & : & \texttt{BOp} \rightarrow V \rightarrow V \rightarrow V \\
\mathcal{O}[\![+]\!]\, \upsilon_1\, \upsilon_2 & = & \upsilon_1 = \epsilon \vee \upsilon_2 = \epsilon \rightarrow \epsilon, (\upsilon_1 + \upsilon_2) \\
\mathcal{O}[\![-]\!]\, \upsilon_1\, \upsilon_2 & = & \upsilon_1 = \epsilon \vee \upsilon_2 = \epsilon \rightarrow \epsilon, (\upsilon_1 - \upsilon_2) \\
\mathcal{O}[\![*]\!]\, \upsilon_1\, \upsilon_2 & = & \upsilon_1 = \epsilon \vee \upsilon_2 = \epsilon \rightarrow \epsilon, (\upsilon_1 \times \upsilon_2) \\
\mathcal{O}[\![\mathrm{DIV}]\!]\, \upsilon_1\, \upsilon_2 & = & \upsilon_1 = \epsilon \vee \upsilon_2 = \epsilon \vee \upsilon_2 = 0 \rightarrow \epsilon, (\upsilon_1 \mid \upsilon_2) \\
\\
\mathcal{E} & : & E \rightarrow S \rightarrow V \\
\mathcal{E}[\![I]\!]\sigma & = & \sigma I \\
\mathcal{E}[\![\mathrm{N}]\!]\sigma & = & \mathcal{N}[\![\mathrm{N}]\!] \\
\mathcal{E}[\![E_1 \ \texttt{BOp} \ E_2]\!]\sigma & = & \mathcal{O}(\mathcal{E}[\![E_1]\!]\sigma)(\mathcal{E}[\![E_2]\!]\sigma) \\
\\
\mathcal{C} & : & C \rightarrow S \rightarrow S \\
\mathcal{C}[\![I := E]\!]\sigma & = & \sigma[I \leftarrow \mathcal{E}[\![E]\!]\sigma] \\
\mathcal{C}[\![C_1 ; C_2]\!] & = & \mathcal{C}[\![C_2]\!] \circ \mathcal{C}[\![C_1]\!] \\
\mathcal{C}[\![\texttt{if}\, E\, \texttt{then}\, C\, \texttt{fi}]\!]\sigma & = & \mathcal{E}[\![E]\!]\sigma = \epsilon \rightarrow \lambda i.\epsilon, \mathcal{E}[\![E]\!]\sigma \neq 0 \rightarrow \mathcal{C}[\![C]\!]\sigma, \sigma \\
\mathcal{C}[\![\texttt{while}\, E\, \texttt{do}\, C\, \texttt{od}]\!] & = & fix(\lambda\omega.\lambda\sigma.\mathcal{E}[\![E]\!]\sigma = \epsilon \rightarrow \lambda i.\epsilon, f\sigma) \\
 & & \text{where } f\sigma = \mathcal{E}[\![E]\!]\sigma \neq 0 \rightarrow \omega(\mathcal{C}[\![C]\!]\sigma), \sigma \\
\\
\sigma[i \leftarrow \upsilon]j & = & \begin{cases} \epsilon & \text{if } \sigma = \lambda x.\epsilon \vee \upsilon = \epsilon \\ \upsilon & \text{if } \sigma \neq \lambda x.\epsilon \wedge \upsilon \neq \epsilon \wedge i = j \\ \sigma(j) & \text{if } \sigma \neq \lambda x.\epsilon \wedge \upsilon \neq \epsilon \wedge i \neq j \end{cases}
\end{array}
$$

Fig. 5. The Strict Semantics (With Respect to Errors) of **L**

$$
\begin{array}{rcl}
\mathcal{C}[\![I := E]\!]\sigma & = & \sigma[I \Leftarrow \mathcal{E}[\![E]\!]\sigma] \\
\mathcal{C}[\![C_1 ; C_2]\!] & = & \mathcal{C}[\![C_2]\!] \circ \mathcal{C}[\![C_1]\!] \\
\mathcal{C}[\![\texttt{if}\, E\, \texttt{then}\, C\, \texttt{fi}]\!]\sigma & = & \mathcal{E}[\![E]\!]\sigma = \epsilon \rightarrow \sigma, \mathcal{E}[\![E]\!]\sigma \neq 0 \rightarrow \mathcal{C}[\![C]\!]\sigma, \sigma \\
\mathcal{C}[\![\texttt{while}\, E\, \texttt{do}\, C\, \texttt{od}]\!] & = & fix(\lambda\omega.\lambda\sigma.\mathcal{E}[\![E]\!]\sigma = \epsilon \rightarrow \sigma, f\sigma) \\
 & & \text{where } f\sigma = \mathcal{E}[\![E]\!]\sigma \neq 0 \rightarrow \omega(\mathcal{C}[\![C]\!]\sigma), \sigma \\
\\
\sigma[i \Leftarrow \upsilon]j & = & \begin{cases} \sigma(j) & \text{if } i \neq j \\ \sigma[i \leftarrow \upsilon]j & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 6. The 'Lazy' Semantics (with Respect to Errors) of **L**

Figure	Strict Meaning	Lazy Meaning
Fig. 1	$\lambda\sigma.\sigma \oplus \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 4, \mathtt{z} \mapsto 2\}$	$\lambda\sigma.\sigma \oplus \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 4, \mathtt{z} \mapsto 2\}$
...Slice	$\lambda\sigma.\sigma \oplus \{\mathtt{x} \mapsto 1, \mathtt{z} \mapsto 2\}$	$\lambda\sigma.\sigma \oplus \{\mathtt{x} \mapsto 1, \mathtt{z} \mapsto 2\}$
Fig. 2	$\lambda\sigma.\epsilon$	$\lambda\sigma.\sigma \oplus \{\mathtt{x} \mapsto \epsilon.\mathtt{y} \mapsto 2\}$
...Slice	$\lambda\sigma.\sigma \oplus \{\mathtt{y} \mapsto 2\}$	$\lambda\sigma.\sigma \oplus \{\mathtt{y} \mapsto 2\}$
Fig. 3	$\lambda\sigma.\epsilon$	$\lambda\sigma.\sigma \oplus \{\mathtt{x} \mapsto \epsilon, \mathtt{z} \mapsto \sigma\mathtt{z}{+}1, \mathtt{y} \mapsto \epsilon\}$
...Slice	$\lambda\sigma.\epsilon$	$\lambda\sigma.\sigma \oplus \{\mathtt{x} \mapsto \epsilon, \mathtt{y} \mapsto \epsilon\}$

Fig. 7. Strict and Lazy Meanings of Programs and Their Slices

$$
\begin{array}{rcl}
\mathcal{D} : C \cup E \rightarrow \mathcal{P}(I) & , & \mathcal{R} : C \cup E \rightarrow \mathcal{P}(I) \\
\\
\mathcal{R}[\![E_1 \ \ \texttt{BOp} \ \ E_2]\!] & = & \mathcal{R}[\![E_1]\!] \cup \mathcal{R}[\![E_2]\!] \\
\mathcal{R}[\![I]\!] & = & \{I\} \\
\mathcal{R}[\![\mathrm{N}]\!] & = & \emptyset \\
\\
\mathcal{R}[\![I := E]\!] & = & \mathcal{R}[\![E]\!] \\
\mathcal{R}[\![C_1;\, C_2]\!] & = & \mathcal{R}[\![C_1]\!] \cup \mathcal{R}[\![C_2]\!] \\
\mathcal{R}[\![\texttt{if}\, E \,\texttt{then}\, C \,\texttt{fi}]\!] & = & \mathcal{R}[\![\texttt{while}\, E \,\texttt{do}\, C \,\texttt{od}]\!] = \mathcal{R}[\![E]\!] \cup \mathcal{R}[\![C]\!] \\
\\
\mathcal{D}[\![E]\!] & = & \emptyset \\
\mathcal{D}[\![I := E]\!] & = & \{I\} \\
\mathcal{D}[\![C_1;\, C_2]\!] & = & \mathcal{D}[\![C_1]\!] \cup \mathcal{D}[\![C_2]\!] \\
\mathcal{D}[\![\texttt{if}\, E \,\texttt{then}\, C \,\texttt{fi}]\!] & = & \mathcal{D}[\![\texttt{while}\, E \,\texttt{do}\, C \,\texttt{od}]\!] = \mathcal{D}[\![C]\!]
\end{array}
$$

Fig. 8. Traditional Algorithm for Defined and Referenced Variables

$$
\begin{array}{rcl}
\mathcal{D}_\epsilon : C \cup E \rightarrow \mathcal{P}(I) & , & \mathcal{R}_\epsilon : C \cup E \rightarrow \mathcal{P}(I) \quad , \quad b \in \ \texttt{BOp} \\
\\
\mathcal{D}_\epsilon[\![E_1 \, b \, E_2]\!] & = & \left\{ \begin{array}{ll} \mathcal{D}_\epsilon[\![E_1]\!] \cup \mathcal{D}_\epsilon[\![E_2]\!] & \text{if } b \in \{+,-,*\} \\ \mathcal{D}_\epsilon[\![E_1]\!] \cup \mathcal{D}_\epsilon[\![E_2]\!] \cup \{Err\} & \text{if } b = \mathrm{DIV} \end{array} \right. \\
\mathcal{D}_\epsilon[\![I]\!] & = & \mathcal{D}_\epsilon[\![\mathrm{N}]\!] = \emptyset \\
\\
\mathcal{R}_\epsilon[\![E_1 \, b \, E_2]\!] & = & \left\{ \begin{array}{ll} \mathcal{R}_\epsilon[\![E_1]\!] \cup \mathcal{R}_\epsilon[\![E_2]\!] & \text{if } b \in \{+,-,*\} \\ \mathcal{R}_\epsilon[\![E_1]\!] \cup \mathcal{R}_\epsilon[\![E_2]\!] \cup \{Err\} & \text{if } b = \mathrm{DIV} \end{array} \right. \\
\\
\mathcal{R}_\epsilon[\![I]\!] & = & \{I\} \\
\mathcal{R}_\epsilon[\![\mathrm{N}]\!] & = & \emptyset \\
\mathcal{R}_\epsilon[\![I := E]\!] & = & \mathcal{R}_\epsilon[\![E]\!] \\
\mathcal{R}_\epsilon[\![C_1;\, C_2]\!] & = & \mathcal{R}_\epsilon[\![C_1]\!] \cup \mathcal{R}_\epsilon[\![C_2]\!] \\
\mathcal{R}_\epsilon[\![\texttt{if}\, E \,\texttt{then}\, C \,\texttt{fi}]\!] & = & \mathcal{R}_\epsilon[\![\texttt{while}\, E \,\texttt{do}\, C \,\texttt{od}]\!] = \mathcal{R}_\epsilon[\![E]\!] \cup \mathcal{R}_\epsilon[\![C]\!] \\
\\
\mathcal{D}_\epsilon[\![I := E]\!] & = & \mathcal{D}_\epsilon[\![E]\!] \cup \{I\} \\
\mathcal{D}_\epsilon[\![C_1;\, C_2]\!] & = & \mathcal{D}_\epsilon[\![C_1]\!] \cup \mathcal{D}_\epsilon[\![C_2]\!] \\
\mathcal{D}_\epsilon[\![\texttt{if}\, E \,\texttt{then}\, C \,\texttt{fi}]\!] & = & \mathcal{D}_\epsilon[\![\texttt{while}\, E \,\texttt{do}\, C \,\texttt{od}]\!] = \mathcal{D}_\epsilon[\![E]\!] \cup \mathcal{D}_\epsilon[\![C]\!]
\end{array}
$$

Fig. 9. Modified Version of Defined and Referenced Variables