Minimal Slicing and the Relationships Between Forms of Slicing

Dave Binkley[1] **Sebastian Danicic**[2] **Tibor Gyimóthy**[3] **Mark Harman**[4] **Ákos Kiss**[3] **Bogdan Korel**[5]

[1]Loyola College
Baltimore MD
21210-2699, USA.

[2]Goldsmiths College
University of London
New Cross, London
SE14 6NW, UK.

[3]Institute of Informatics
University of Szeged
6720 Szeged, Hungary.

[4]King's College London
Strand, London
WC2R 2LS, UK.

[5]Illinois Institute of Technology
Chicago IL
60616-3793, USA.

Abstract

The widespread interest in program slicing within the source code analysis and manipulation community has led to the introduction of a large number of different slicing techniques. Each preserves some aspect of a program's behaviour and simplifies the program to focus exclusively upon this behaviour. In order to understand the similarities and differences between slicing techniques, a formal mechanism is required. This paper establishes a formal mechanism for comparing slicing techniques using a theory of program projection. Sets of minimal slices, which form the ideal for any slicing algorithm, are used to reveal the ordering relationship between various static and dynamic slicing techniques.

1. Introduction

Program slicing has been the subject of widespread study in the literature on source code analysis and manipulation. Much of this previous work has considered algorithmic details and the nature of dependence for the particular slicing technique chosen (*e.g.*, static backward slicing, or forward dynamic slicing). As such, previous work has tended to provide intra-technique study; focusing exclusively upon one form of slicing technique and the relationships it creates among code level components. By contrast, this paper is concerned with the meta-level relationships between slicing techniques. That is, the paper studies inter-technique relationships.

By focusing on inter-technique relationships, we can ask questions about which slicing techniques are intrinsically superior, in the sense that the slices according to one are all slices according to the other. This relationship is called the subsumes relationship between slicing techniques. Our previous work [2] formalised this inter-technique relationship for static and dynamic slicing, revealing that the subsumes relationship was more subtle and intricate than previously thought.

This paper aims to formalize the other fundamental inter-technique slicing relationship: the rank relationship. At the level of individual slices, the rank relationship is vital: it determines whether one slice is better than (that is smaller than) another. In all applications of slicing, the size of slices is crucial: the smaller the better. At the meta level of inter-technique relationships, the rank relationship allows us to determine whether one definition of slicing leads to inherently smaller slices than another. This puts statements such as

> "dynamic slices are smaller than static slices"

on a firm theoretical footing. We know intuitively what we mean by such statements, but capturing this formally is non-trivial. Clearly, not all dynamic slices are smaller than all static slices. Even for a given choice of program point and variable, the statement may not be true, because of differences in slicing algorithm. Furthermore, there is the complication of which particular dynamic slicing definition one is to adopt; some are incomparable with static slicing.

This paper addresses all of these issues, giving a formal mechanism for investigation of the rank relationship among slicing techniques. It uses this to reveal the *rank lattice* which depicts the rank relationships for various dynamic and static slicing techniques. The primary technical contributions are the following.

1. The formalization of a ranking of slicing techniques.
2. The two principal inter-technique relationships of subsumption and rank are shown to be duals of one another.
3. Using this duality a lattice of rank relationships for static and dynamic slicing techniques is constructed.

The paper makes a contribution to a larger goal: the formalization of the theoretical foundations of program slicing.

Clearly such a formalization requires more work than can reasonably be presented in a single article; the present paper makes only a small contribution to this overall goal. However, the authors believe that the practice of slicing is now sufficiently mature and stable to warrant a unifying formal theory to underpin further developments and to consolidate existing practice. To stimulate interest in this larger, high level, aim the paper sets out a 'manifesto' (or programme of work) needed to provide a formal theory of slicing.

The rest of the paper is organised as follows. Section 2 sets out a manifesto for a research programme in formalizing the theoretical foundations of program slicing. This paper then goes on to make a contribution to this manifesto. Sections 3 summarise our previous work on the program projection framework and the subsumes relationship in order to make the present paper self-contained. Section 4 presents the first key technical results dealing with sets of minimal slices. This result is then leveraged in Section 5 to formally define the rank relationship for slicing techniques and to reveal the lattice of rank relationships between static and dynamic slicing techniques. Section 6 briefly presents related work and Section 7 concludes.

2. A Manifesto

The program projection theory provides a general framework, within which many current approaches to slicing could be formulated. In this section the authors set out a seven point research agenda for a theoretical formalization of slicing with the aim of stimulating interest in the wider research community in tackling some of these problems.

1. **Criteria**
   This paper shows how dynamic forms of slicing can be brought within the projection theory framework, allowing relationships between them to be explored. There are many other forms of slicing, which might benefit from a similar treatment. For example, Conditioned [5], decomposition [11], quasi-static [24], forward [17] and amorphous [15] forms of slicing. There has been some work in this area, for example amorphous static slicing has been expressed within the projection framework [13], but more work is required to formulate all forms of slicing within the projection framework.

2. **Minimality**
   Much work needs to be done to characterise and understand the forms of program that have minimal slices for each kind of slicing criterion. The theoretical results which would accrue form this research endeavor would have far–reaching implications for the applications of slicing, since the size of the slice is important in all applications. Recent work [9] has shown that program schematology may be useful in the formulation and investigation of questions of slice minimality.

3. **Complexity**
   The algorithmic complexity of static, syntax preserving, slicing has been well-known for some time [17, 21], but for other forms of slicing, the bounds on algorithmic complexity are less well understood.

4. **Notation**
   Although a comparatively trivial problem, the issue of a unified notation for expressing notions of slicing and their slicing criteria remains important. A common, widely used and unified notation would help to facilitate communication and would assist in expressing the relationships between forms of slicing.

5. **Semantics**
   Slicing does not preserve the traditional strict semantics of the programming language in which subject programs and their slices are expressed [6, 16]. It is therefore necessary to define and capture the semantics preserved by slicing algorithms. This is a form of theory 'reverse engineering', since the algorithms are already in place, and the theory is dragging somewhat behind. Abstract Interpretation [7] would clearly be useful in this part of the research programme. An understanding of the semantics preserved by slicing is crucial to provonmg correctness. Hitherto, only intraprocedural static, syntax-preserving slcing has been proved correct [22].

6. **Relationships**
   This paper shows how formalization of slicing criteria within the projection framework allows for slices to be compared using a lattice theoretic approach. The authors have studied two forms[1] of subsumes relationship between slicing criteria, but there may be other interesting formulations of slicing criteria subsumption. The ability to explore the relationships between slicing criteria is one of the principal benefits of the theory we advocate. The results should help us to better understand slicing criteria.

7. **Executability**
   Some forms and slicing are executable. These are easier to fit into a theoretical framework, such as that proposed in the present paper, because there is an equivalence between the slice and the original program. However, there are non-executable forms of program slice [24], and it remains an open challenge as to how these can be defined formally and compared to execrable forms of slicing.

[1] In the present paper and a previous paper [2].

3. Program Projection Theory

The *projection theory* is, in essence, a generalization of program slicing [14, 13]. It is defined with respect to two relations on programs: a *syntactic ordering* and a *semantic equivalence*. The syntactic ordering is simply an ordering relation on programs. It is used to capture the syntactic property that slicing seeks to optimize. Programs that are lower according to the ordering are considered to be 'better'. The semantic relation is an equivalence relation that captures the semantic property that remains invariant during slicing.

Definition 1 (Syntactic Ordering) A syntactic ordering, denoted by $\lesssim$, is a computable partial order on programs.

Definition 2 (Semantic Equivalence) A semantic equivalence, denoted by $\approx$, is an equivalence relation on program semantics.

Definition 3 (($\lesssim, \approx$) Projection) Given syntactic ordering $\lesssim$ and semantic equivalence $\approx$,

$$\text{program } p \text{ is a } (\lesssim, \approx) \textit{ projection} \text{ of program } q \\ \text{iff} \\ p \lesssim q \;\wedge\; p \approx q.$$

That is, in a projection, the syntax can only improve while the semantics of interest must remain unchanged. Notice that a program may have several projections for a given syntactic ordering and semantic equivalence, so we will use the following notation to represent all of them:

Definition 4 (Set of All Possible Slices) The set of all possible slices of a program p for a given projection $(\lesssim, \approx)$ is defined as follows:

$$\mathbb{S}_p(\lesssim, \approx) = \{q | q \approx p \text{ and } q \lesssim p\}.$$

In this paper, we consider only syntax-preserving forms of slicing [13]. Therefore, we will use the following syntactic ordering, which formalizes the oft-quoted remark: "a slice is a subset of the program from which it is constructed". Note that for ease of presentation, it is assumed that each program component occupies a unique line. Thus, a line number can be used to uniquely identify a particular program component.

Definition 5 (Traditional Syntactic Ordering [13]) Let F be a function that takes a program and returns a partial function from line-numbers to statements, such that the function $F(p)$ maps l to c iff program p contains the statement c at line number l. The syntactic ordering, denoted by $\sqsubseteq$, is defined as follows:

$$p \sqsubseteq q \Leftrightarrow F(p) \subseteq F(q).$$

The semantic property that various slicing techniques respect is based upon the concept of a *state trajectory*. The following definitions of *state trajectory* and *state restriction* are extracted from Weiser's definition of a slice [26].

Definition 6 (State Trajectory) A *state trajectory* is a finite sequence of line-number $\times$ state pairs:

$$(n_1, \sigma_1)(n_2, \sigma_2) \ldots (n_k, \sigma_k),$$

where a state is a partial function mapping a variable to a value, and entry i is (n_i, σ_i) if after i statement executions the state is σ_i, and the next instruction to be executed is at line number n_i.

Definition 7 (State Restriction) Given a state, σ and a set of variables V, $\sigma \mid V$ restricts σ so that it is defined only for variables in V:

$$(\sigma \mid V)x = \begin{cases} \sigma\, x & \text{if } x \in V, \\ \perp & \text{otherwise.} \end{cases}$$

The two auxiliary functions defined below, $Proj'^*$ and $Proj^*$, are the counter parts of Weiser's $Proj'$ and $Proj$, extended to be able to describe not only static slicing but other slicing methods, such as Korel and Laski's dynamic slicing. $Proj^*$, with the help of $Proj'^*$, extracts from a state trajectory the values of the variables at the point of interest and, additionally, retains information about the execution path. Since not only static slicing is considered, the point of interest is not only n, a line number, but n^k, the k^{th} occurrence of instruction n in the trajectory.

Definition 8 ($\boldsymbol{Proj'^*}$) $Proj'^*$ is defined in terms of 5 parameters: a set of variables V, a set of line-number $\times$ natural number pairs P, a set of line numbers I, a line-number $\times$ natural number pair n^k, and a state σ:

$$Proj'^*_{(V,P,I)}(n^k, \sigma) = \begin{cases} (n, \sigma|V) & \text{if } n^k \in P, \\ (n, \perp) & \text{if } n^k \notin P \text{ and } n \in I, \\ \lambda & \text{otherwise.} \end{cases}$$

Definition 9 ($\boldsymbol{Proj^*}$) For a set of variables V, set of line-number $\times$ natural number pairs P, set of line numbers I and trajectory T:

$$Proj^*_{(V,P,I)}(T) = \prod_{i=1}^{l} Proj'^*_{(V,P,I)}({n_i}^{k_i}, \sigma_i),$$

where k_i is the number of occurrences of n_i in the first i elements of T (*i.e.*, ${n_i}^{k_i}$ is the most recent occurrence of n_i in $T[0] \ldots T[i]$), and l is the highest index in T such that ${n_l}^{k_l} \in P$.

Using the above functions we can define a unified semantic equivalence relation $\mathcal{U}$ capable of expressing multiple slicing techniques.

Definition 10 (Unified Equivalence) Given programs p and q, a set of states S, a set of variables V, a set of line-number $\times$ natural number pairs P, and a set of line-numbers $\times$ set of line-numbers $\rightarrow$ set of line-numbers function X, the unified equivalence ($\mathcal{U}$) is defined as follows:

$$p \quad \mathcal{U}(S, V, P, X) \quad q$$
$$\text{iff}$$
$$\forall \sigma \in S : Proj^*_{(V,P,X(\bar{p},\bar{q}))}(T^\sigma_p) = Proj^*_{(V,P,X(\bar{p},\bar{q}))}(T^\sigma_q).$$

In the above definition, the roles of the parameters are as follows: S denotes the set of initial states for which the equivalence must hold. This captures the 'input' part of the slicing criteria. The set of variables of interest V is common to all slicing criteria. Parameter P, just as in Definitions 8 and 9, contains the points of interest in the trajectory and also captures the 'iteration count' component of the criteria. Finally, X captures the 'trajectory requirement'. It is a function that determines which statements must be preserved in the trajectory (even though they have no affect on the variables of the slicing technique). The domain of X is a pair of sets of statement numbers from two programs. For program p, the set of statement numbers is denoted as $\bar{p}$.

This definition can be instantiated with different parameters. The following eight equivalence relations use $\mathbf{\Sigma}$ to denote the set of all possible input states, and $\mathbf{N}$ to denote the set of natural numbers (thus $\{n\} \times \mathbf{N}$ represents all occurrences of instruction n). For every set of line numbers, x and y, $E(x, y) = \emptyset$, and $\cap$ denotes the set intersection operation.

$$\begin{array}{lcl}
\mathcal{S}(V, n) & = & \mathcal{U}(\mathbf{\Sigma}, V, \{n\} \times \mathbf{N}, E), \\
\mathcal{S}_i(V, n^k) & = & \mathcal{U}(\mathbf{\Sigma}, V, \{n^k\}, E), \\
\mathcal{D}(\sigma, V, n) & = & \mathcal{U}(\{\sigma\}, V, \{n\} \times \mathbf{N}, E), \\
\mathcal{D}_i(\sigma, V, n^k) & = & \mathcal{U}(\{\sigma\}, V, \{n^k\}, E), \\
\mathcal{S}_{KL}(V, n) & = & \mathcal{U}(\mathbf{\Sigma}, V, \{n\} \times \mathbf{N}, \cap), \\
\mathcal{S}_{KLi}(V, n^k) & = & \mathcal{U}(\mathbf{\Sigma}, V, \{n^k\}, \cap), \\
\mathcal{D}_{KL}(\sigma, V, n) & = & \mathcal{U}(\{\sigma\}, V, \{n\} \times \mathbf{N}, \cap), \\
\mathcal{D}_{KLi}(\sigma, V, n^k) & = & \mathcal{U}(\{\sigma\}, V, \{n^k\}, \cap).
\end{array}$$

Two of the eight equivalence relations defined above, $\mathcal{S}(V, n)$ and $\mathcal{D}_{KLi}(\sigma, V, n^k)$ capture the semantics of Weiser's static and Korel and Laski's dynamic slicing. The other six equivalence relations result from other possible parameterizations of the unified equivalence.

4. Sets of Minimal Slices

To compare slicing techniques, it is important to be free from algorithmic and implementation details. We are concerned with the investigation of various definitions for 'slice'; not the peculiarities which emerge from attempts to arrive at 'good' slicing algorithms. In other words, the output of idealized algorithms are studied. Any realizable slicing algorithm must by definition, compute an *approximation* to the idealized algorithm. Even with idealized algorithms there is no guarantee of a unique minimal slice. Therefore, sets of minimal slices are studied. Such a set includes all the 'best' (*i.e.*, smallest) slices.

To formalize the beliefs about the size of slices (more precisely, about the size of minimal slices), we shall compare sets of minimal slices. To allow such comparison, we have to extend the syntactic ordering of programs (from Definition 1) to sets of programs. The extension is, in fact, more general than required. It addresses sets of incomparable slices. Of interest in Section 5 are sets of minimal slices, which are also sets of incomparable slices.

In the following, we will use the notation $a \not\gtrsim\not\lesssim b$ to denote that a is not comparable to b in the partial order $\lesssim$, *i.e.*, $a \not\lesssim b$ and $b \not\lesssim a$.

Definition 11 (Set of Incomparable Programs) A is a set of incomparable programs with respect to syntactic ordering $\lesssim$, iff

$$\forall a, b \in A : a \not\gtrsim\not\lesssim b.$$

Definition 12 (Syntactic Ordering of Sets of Incomparable Programs) For syntactic ordering $\lesssim$ and two sets of incomparable programs with respect to $\lesssim$, A and B,

$$A \lessapprox B$$
$$\text{iff}$$
$$\forall b \in B : \exists a \in A : a \lesssim b$$
$$\text{and}$$
$$\forall b \in B : \nexists a \in A : b \lnsim a.$$

Before using sets of incomparable slices in 5 it is first necessary to show that the syntactic ordering from Definition 12 is a partial order.

Theorem 1 The syntactic ordering is a partial order over sets of incomparable programs.

Proof

We have to show that the relation given in Definition 12 is reflexive, antisymmetric, and transitive.

Reflexivity. We have to show that $A \lessapprox A$ holds for all sets of incomparable programs. According to the definition this involves two things. First, we have to show that $\forall a \in A : \exists a' \in A : a' \lesssim a$. This follows as $a \lesssim a$. Second, we have to show that $\forall a' \in A : \nexists a \in A : a' \lnsim a$. Because all elements in A are incomparable with respect to $\lesssim$, no such a' exists.

Antisymmetry. We have to show for all sets of incomparable programs A and B, $A \lessapprox B$ and $B \lessapprox A$ implies $A = B$. From $A \lessapprox B$, we know that $\forall b \in B : \exists a \in A : a \lesssim b$ and from $B \lessapprox A$ we know that $\forall a \in A : \exists b' \in B : b' \lesssim a$. Together, this yields $b' \lesssim a \lesssim b$. However, as all elements in B are incomparable, $b' = b$; thus, $b \lesssim a \lesssim b$ which implies that $a = b$. Consequently, $\forall b \in B : b \in A$, *i.e.*, $B \subseteq A$. Finally, by symmetry, $A \subseteq B$, from which follows that $A = B$, as required.

Transitivity. We have to show for all A, B and C sets of incomparable programs that $A \lessapprox B$ and $B \lessapprox C$ imply $A \lessapprox C$. This involves showing two things: first, that $\forall c \in C : \exists a \in A : a \lesssim c$. From $B \lessapprox C$ we know that $\forall c \in C : \exists b \in B : b \lesssim c$. Furthermore, from $A \lessapprox B$ we know that $\forall b \in B : \exists a \in A : a \lesssim b$. Together these imply $\forall c \in C : \exists a \in A : a \lesssim c$ as required.

Second, we must show that $\forall c \in C : \nexists a \in A : c \lnsim a$, which is equivalent to $\forall c \in C : \forall a \in A : a \lesssim c$ or $a \not\gtrsim\not\lesssim c$. (Note that the negation of $c \lnsim a$ is $a \lesssim c$ or $a \not\gtrsim\not\lesssim c$). By assumption $\forall b \in B : \forall a \in A : a \lesssim b$ or $a \not\gtrsim\not\lesssim b$, and $\forall c \in C : \forall b \in B : b \lesssim c$ or $b \not\gtrsim\not\lesssim c$. This leads to the following four cases.

(Case 1) $b \lesssim c$ **and** $a \lesssim b$**:** Here trivially $a \lesssim c$.

(Case 2) $b \lesssim c$ **but** $a \not\gtrsim\not\lesssim b$**:** We prove Case 2 by contradiction. Assume, that ($a \lesssim c$ or $a \not\gtrsim\not\lesssim c$) is not true. This assumption simplifies to $c \lnsim a$. Combined with $b \lesssim c$ this yields $b \lesssim c \lnsim a$ from which it follows that $b \lnsim a$. However, this contradicts the assumption $a \not\gtrsim\not\lesssim b$.

(Case 3) $b \not\gtrsim\not\lesssim c$ **and** $a \lesssim b$**:** We prove Case 3 by contradiction. Assume as in Case 2, $c \lnsim a$. Combined with $a \lesssim b$ yields $c \lnsim a \lesssim b$ from which it follows that $c \lnsim b$. This contradicts the assumption $b \not\gtrsim\not\lesssim c$.

(Case 4) $a \not\gtrsim\not\lesssim b$ **and** $b \not\gtrsim\not\lesssim c$**:** Once again, assume $c \lnsim a$. By assumption know that $\exists b' \in B : b' \lesssim c$ and $\exists a' \in A : a' \lesssim b'$. However, $a' \lesssim b' \lesssim c \lnsim a$ implies $a' \lnsim a$, which contradicts the assumption that the elements in A are incomparable.

□

One might think that there shall be more natural extensions of the syntactic ordering to the domain of incomparable program sets than that given in Definition 12. We could, for example, define A less than B iff all elements of A are less than all elements of B. Notice, however, that this definition is not a partial order. Assume that $A = \{a_1, a_2\}$, where $a_1 \not\gtrsim\not\lesssim a_2$. This implies, according to the hypothetical definition above, that $A \lessapprox A$ is not true, thus reflexivity is broken. This shows that we cannot have a too strong requirement on comparability. We shall allow some elements to be incomparable so long as there is one element which is comparable. Definition 12 captures the right balance in this field, and is still an effective extension of the syntactic ordering, since if given two one-element sets, $A = \{a\}$ and $B = \{b\}$, then $A \lessapprox B$ iff $a \lesssim b$.

Now that we have all the necessary definitions we can turn to minimal slices. Since minimal slices are not necessarily unique, we work with sets of minimal slices, which are formally defined below.

Definition 13 (Set of All Minimal Slices) The set of all minimal slices of a program p for a given projection $(\lesssim, \approx)$, denoted by $\mathbb{M}_p(\lesssim, \approx)$, is defined as follows:

$$\mathbb{M}_p(\lesssim, \approx) =$$
$$\{q | q \in \mathbb{S}_p(\lesssim, \approx) \text{ and } \nexists q' \in \mathbb{S}_p(\lesssim, \approx) : q' \lnsim q\}.$$

Inspecting the above definition, we can notice that the set of all minimal slices of a program is a set of incomparable programs (as defined in Definition 11) and thus we can use the extended syntactic order relation on them. Below we state the central theorem regarding the connection between the sets of slices and sets of minimal slices. Informally, given a program, if its slices for projection A are valid slices for projection B as well, then the minimal slices for B are smaller than the minimal slices for A.

Theorem 2 (Duality of Slices) For any program p, syntactic ordering $\lesssim$, and semantic equivalence relations $\approx_A$ and $\approx_B$ the following holds:

$$\mathbb{S}_p(\lesssim, \approx_A) \subseteq \mathbb{S}_p(\lesssim, \approx_B) \Rightarrow \mathbb{M}_p(\lesssim, \approx_B) \lessapprox \mathbb{M}_p(\lesssim, \approx_A).$$

Proof

Observe that if $a \in \mathbb{M}_p(\lesssim, \approx_A)$ then, by definition, $a \in \mathbb{S}_p(\lesssim, \approx_A)$, and also $a \in \mathbb{S}_p(\lesssim, \approx_B)$, as $\mathbb{S}_p(\lesssim, \approx_A) \subseteq \mathbb{S}_p(\lesssim, \approx_B)$. We have two cases to consider.

(Case 1): We want to show that

$$\forall a \in \mathbb{M}_p(\lesssim, \approx_A) : \exists b \in \mathbb{M}_p(\lesssim, \approx_B) : b \lesssim a.$$

If $a \in \mathbb{M}_p(\lesssim, \approx_B)$, then we are done as $a \lesssim a$, otherwise (if $a \notin \mathbb{M}_p(\lesssim, \approx_B)$), then, by definition, $\exists b' \in \mathbb{M}_p(\lesssim, \approx_B) : b' \lnsim a$. In this case, let $b = b'$.

(Case 2): The second case is to show that

$$\forall a \in \mathbb{M}_p(\lesssim, \approx_A) : \nexists b \in \mathbb{M}_p(\lesssim, \approx_B) : a \lnsim b.$$

Let $a \in \mathbb{M}_p(\lesssim, \approx_A)$ and assume that $b \in \mathbb{M}_p(\lesssim, \approx_B) : a \lnsim b$. This contradicts $\mathbb{M}_p(\lesssim, \approx_B)$ being a set of minimal slices as $a \in \mathbb{S}_p(\lesssim, \approx_B)$ would have to be in $\mathbb{M}_p(\lesssim, \approx_B)$, but then b would not be in $\mathbb{M}_p(\lesssim, \approx_B)$ as $a \lnsim b$.

□

Program p	q_1	q_2	q_3
`x=1;`			
`x=2;`			
`if (x>1)`			
`  y=1;`	`  y=1;`		
`else`			
`  y=1;`		`  y=1;`	
`scanf("%d",&x);`	`scanf("%d",&x);`	`scanf("%d",&x);`	
`if (x<1)`	`if (x<1)`	`if (x<1)`	
`  z=0;`	`  z=0;`	`  z=0;`	`z=0;`
`else`	`else`	`else`	
`  z=x*y;`	`  z=x*y;`	`  z=x*y;`	
`w=z;`	`w=z;`	`w=z;`	`w=z;`

$\sigma = \langle 0 \rangle, V = \{\mathtt{z}\}, n = 12, k = 1$

Figure 1. Example program to show that the reverse of the duality theorem is not true.

Interestingly, the reverse of Theorem 2 is not true, *i.e.*, $\mathbb{M}_p(\lesssim, \approx_B) \lesssim \mathbb{M}_p(\lesssim, \approx_A)$ does not imply $\mathbb{S}_p(\lesssim, \approx_A) \subseteq \mathbb{S}_p(\lesssim, \approx_B)$. As a counter example, consider program p in Figure 1. In this case there are two minimal static slices, *i.e.*, $\mathbb{M}_p(\sqsubseteq, \mathcal{S}^{(V,n)}) = \{q_1, q_2\}$, while the set of minimal (Korel and Laski–style) dynamic slices is of only one element, $\mathbb{M}_p(\sqsubseteq, \mathcal{D}_{KLi}^{(\sigma,V,n^k)}) = \{q_3\}$. Clearly, $\mathbb{M}_p(\sqsubseteq, \mathcal{D}_{KLi}^{(\sigma,V,n^k)}) \sqsubseteq \mathbb{M}_p(\sqsubseteq, \mathcal{S}^{(V,n)})$, since $q_3 \sqsubseteq q_1$ and $q_3 \sqsubseteq q_2$, but $\mathbb{S}_p(\sqsubseteq, \mathcal{S}^{(V,n)}) \not\subseteq \mathbb{S}_p(\sqsubseteq, \mathcal{D}_{KLi}^{(\sigma,V,n^k)})$, since $q_2 \notin \mathbb{S}_p(\sqsubseteq, \mathcal{D}_{KLi}^{(\sigma,V,n^k)})$.

5 Slicing Techniques

Section 4 provides the basis for the comparison of slicing techniques and sets of related techniques, such as static slicing techniques or dynamic slicing techniques. It provides the necessary machinery to explore the inter-technique relationship: rank and to formalize observations such as 'dynamic slices are smaller than static slices'. This is done for all possible programs and all possible slicing criteria admissible to a chosen technique.

To facilitate this exploration, we extend the semantic relation and syntactic ordering from Section 4 to apply to *slicing techniques*. A slicing technique is essentially a projection, $(\lesssim, \approx)$, in which the syntactic ordering, $\lesssim$, is unchanged same, but the semantic equivalence, $\approx$, is *parameterized* by four values. These four parameters are sufficiently general to capture the range of possible slicing techniques studied. More specifically, the four parameters that denote a slicing criterion are (σ, V, n, k), where V is the set of variables of interest at the k^{th} iteration of instruction n, for input σ. Not all techniques (equivalence relation classes) use all four components, *e.g.*, a static slicing criterion composes only of V and n. On the contrary, dynamic slicing makes use of all four criterion components.

As introduced in Section 3, our previous work formally defined eight parameterized equivalence relations and eight corresponding slicing techniques [2]. It has also established a lattice which describes the subsumption relation between them. In order to make the present paper self-contained we repeat the subsumption lattice in Figure 2 and the definition of subsumption, which informally states that all "B" slices are also "A" slices:

Definition 14 (Subsumes Relation of Slicing Techniques) Given syntactic ordering $\lesssim$ and semantic equivalence relations $\approx_A$ and $\approx_B$, both parameterized by σ, V, n and k, $(\lesssim, \approx_A)$–slicing subsumes $(\lesssim, \approx_B)$–slicing, denoted as $(\lesssim, \approx_B) \subseteq (\lesssim, \approx_A)$, iff

$$\forall p, \sigma, V, n, k : \mathbb{S}_p(\lesssim, \approx_B^{(\sigma,V,n,k)}) \subseteq \mathbb{S}_p(\lesssim, \approx_A^{(\sigma,V,n,k)}).$$

To rank techniques (recall that the rank relationship allows us to determine whether one definition of slicing leads to inherently smaller slices than another) we extend the syntactic ordering definition to slicing techniques (parallel to the extension of Definition 14) and then show that the duality exists. Moreover, in Figure 3 we show that the rank ordering of existing slicing techniques results in a lattice isomorphic (in this case inverted) to that given in Figure 2. This captures the duality of the relationship between subsumption and rank. The formalization has two parts that are captured by Theorem 3 and 4.

Definition 15 (Rank Ordering of Slicing Techniques) For any two slicing techniques, $(\lesssim, \approx_A)$ and $(\lesssim, \approx_B)$,

$$(\lesssim, \approx_A) \lesssim (\lesssim, \approx_B)$$
$$\text{iff}$$
$$\forall p, \sigma, V, n, k : \mathbb{M}_p(\lesssim, \approx_A^{(\sigma,V,n,k)}) \lesssim \mathbb{M}_p(\lesssim, \approx_B^{(\sigma,V,n,k)}).$$

Theorem 3 (Duality of Slicing Techniques) For any two slicing techniques, $(\lesssim, \approx_A)$ and $(\lesssim, \approx_B)$,

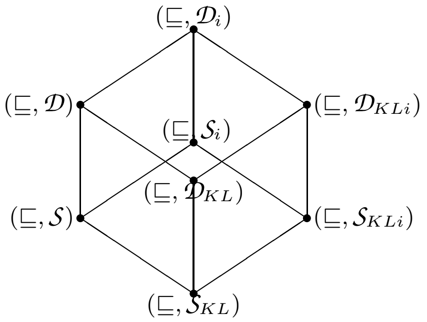

Figure 2. Slicing techniques ordered by the subsumes relation as defined in Defintion 14.

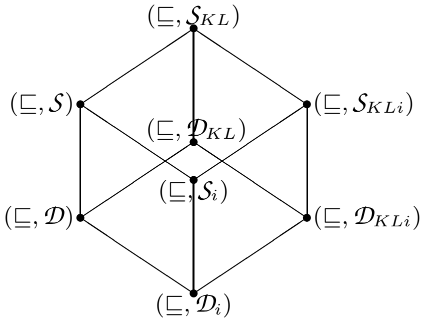

Figure 3. Slicing techniques ordered by rank as defined in Defintion 15.

$$(\lesssim, \approx_A) \subseteq (\lesssim, \approx_B) \Rightarrow (\lesssim, \approx_B) \lesssim (\lesssim, \approx_A).$$

Proof

According to Definition 14, $(\lesssim, \approx_A) \subseteq (\lesssim, \approx_B)$ means $\forall p, \sigma, V, n, k \ : \ \mathbb{S}_p(\lesssim, \approx_A^{(\sigma,V,n,k)}) \ \subseteq \ \mathbb{S}_p(\lesssim, \approx_B^{(\sigma,V,n,k)})$. Theorem 2 proved that $\forall p, \sigma, V, n, k \ : \ \mathbb{M}_p(\lesssim, \approx_B^{(\sigma,V,n,k)}) \lesssim \mathbb{M}_p(\lesssim, \approx_A^{(\sigma,V,n,k)})$, which, by Definition 15, is equivalent to $(\lesssim, \approx_B) \lesssim (\lesssim, \approx_A)$. □

This theorem establishes that if technique A subsumes technique B then B has lower rank (the minimal slices of B will be less than those of A). That is, A will tend to produce larger slices. However, $(\lesssim, \approx_A) \not\subseteq (\lesssim, \approx_B)$ does not imply that $(\lesssim, \approx_B) \not\lesssim (\lesssim, \approx_A)$; thus, it must be shown that those slicing techniques not connected in Figure 3 are really not related according to the traditional rank ordering. This is the role of the following theorem.

`1 x=1;`
`2 x=2;`
`3 if (x>1)`
`4   y=1;`
`5 else`
`6   y=1;`
`7 z=y;`
	`4   y=1;`
	`7 z=y;`
		`6   y=1;`
		`7 z=y;`

Program p	q_1	q_2

$\sigma = \langle\rangle, V = \{\mathtt{y}\}, n = 7, k = 1$

Figure 4. Non–KL (execution path unaware) minimal slices.

Theorem 4 (Duality of Slicing Techniques (only if))

If two slicing techniques are not connected in Figure 3 then they are not related according to the traditional syntactic ordering.

Proof

For each unconnected pair of slicing techniques, $(\sqsubseteq, \approx_A)$ and $(\sqsubseteq, \approx_B)$, we have to show that $(\sqsubseteq, \approx_A) \not\sqsupseteq\not\sqsubseteq (\sqsubseteq, \approx_B)$, in other words that

$$\exists p, \sigma, V, n, k :$$
$$\mathbb{M}_p(\sqsubseteq, \approx_A^{(\sigma,V,n,k)}) \not\sqsubseteq \mathbb{M}_p(\lesssim, \approx_B^{(\sigma,V,n,k)})$$

and

$$\exists p', \sigma', V', n', k' :$$
$$\mathbb{M}_{p'}(\sqsubseteq, \approx_B^{(\sigma',V',n',k')}) \not\sqsubseteq \mathbb{M}_{p'}(\sqsubseteq, \approx_A^{(\sigma',V',n',k')}).$$

The proof makes use of the three counter examples shown in Figures 4, 5, and 6. Implication from these counter examples are combined, as shown in Figure 7, to prove incomparable the pairs of slicing techniques that go unconnected in Figure 3.

First, we show that execution path aware (Korel and Laski–style) slicing techniques, denoted by a subscript KL, are not smaller than execution path unaware (or non–Korel and Laski–style) ones, denoted by a subscript $\cancel{KL}$ Figure 4 gives p, σ, V, n and k, while the two equations below give the sets of minimal slices for each slicing technique.

$$\begin{aligned} M_{KL} &= \{q_1\} \\ &= \mathbb{M}_p(\sqsubseteq, \mathcal{S}_{KL}(V, n)) \\ &= \mathbb{M}_p(\sqsubseteq, \mathcal{S}_{KLi}(V, n^k)) \\ &= \mathbb{M}_p(\sqsubseteq, \mathcal{D}_{KL}(\sigma, V, n)) \\ &= \mathbb{M}_p(\sqsubseteq, \mathcal{D}_{KLi}(\sigma, V, n^k)) \end{aligned}$$

$$\begin{aligned} M_{\cancel{KL}} &= \{q_1, q_2\} \\ &= \mathbb{M}_p(\sqsubseteq, \mathcal{S}(V, n)) \\ &= \mathbb{M}_p(\sqsubseteq, \mathcal{S}_i(V, n^k)) \\ &= \mathbb{M}_p(\sqsubseteq, \mathcal{D}(\sigma, V, n)) \\ &= \mathbb{M}_p(\sqsubseteq, \mathcal{D}_i(\sigma, V, n^k)) \end{aligned}$$

Program p'	Program q'
`x=1;`	`x=1;`
`while (x<=2) {`	`while (x<=2) {`
`  y=1;`	`  y=1;`
`  if (x==1)`	
`    y=2;`	
`  z=y;`	`  z=y;`
`  x++;`	
`}`	`}`

$\sigma' = \langle\rangle, V' = \{\mathtt{y}\}, n' = 6, k' = 2$

Figure 5. Iteration count aware minimal slice.

Program p''	Program q''
`y=1;`	`y=1;`
`scanf("%d",&x);`	
`if (x>1)`	
`  y=2;`	
`z=y;`	`z=y;`

$\sigma'' = \langle 1\rangle, V'' = \{\mathtt{y}\}, n'' = 5, k'' = 1$

Figure 6. Dynamic minimal slice.

From this follows by definition that $M_{KL} \not\sqsubseteq M_{\cancel{KL}}$, since $\nexists q \in M_{KL} : q \sqsubseteq q_2 (\in M_{\cancel{KL}})$.

Now we prove that iteration count unaware slicing techniques are not smaller than iteration count aware ones. Figure 5 and the two equations below give p', σ', V', n' and k', and the minimal slice sets for the slicing techniques.

$$
\begin{aligned}
M_i &= \{q'\}\\
&= \mathbb{M}_{p'}(\sqsubseteq, \mathcal{S}_i(V', {n'}^{k'}))\\
&= \mathbb{M}_{p'}(\sqsubseteq, \mathcal{S}_{KLi}(V', {n'}^{k'}))\\
&= \mathbb{M}_{p'}(\sqsubseteq, \mathcal{D}_i(\sigma', V', {n'}^{k'}))\\
&= \mathbb{M}_{p'}(\sqsubseteq, \mathcal{D}_{KLi}(\sigma', V', {n'}^{k'}))
\end{aligned}
$$

$$
\begin{aligned}
M_{\cancel{i}} &= \{p'\}\\
&= \mathbb{M}_{p'}(\sqsubseteq, \mathcal{S}(V', n'))\\
&= \mathbb{M}_{p'}(\sqsubseteq, \mathcal{S}_{KL}(V', n'))\\
&= \mathbb{M}_{p'}(\sqsubseteq, \mathcal{D}(\sigma', V', n'))\\
&= \mathbb{M}_{p'}(\sqsubseteq, \mathcal{D}_{KL}(\sigma', V', n'))
\end{aligned}
$$

Again, by definition, the above equations imply that $M_{\cancel{i}} \not\sqsubseteq M_i$, since $q'(\in M_i) \sqsubset p'(\in M_{\cancel{i}})$.

Finally, we show that static slicing techniques are not smaller than dynamic slicing techniques. In Figure 6 p'', σ'', V'', n'' and k'' is given and below the sets of minimal slices are computed for all slicing techniques.

Incomparability	Follows from
$(\sqsubseteq, \mathcal{D}) \not\sqsupseteq\not\sqsubseteq (\sqsubseteq, \mathcal{D}_{KLi})$	$M_{KL} \not\sqsubseteq M_{\cancel{KL}}$ and $M_{\cancel{i}} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{D}) \not\sqsupseteq\not\sqsubseteq (\sqsubseteq, \mathcal{S}_i)$	$M_S \not\sqsubseteq M_D$ and $M_{\cancel{i}} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{D}_{KLi}) \not\sqsupseteq\not\sqsubseteq (\sqsubseteq, \mathcal{S}_i)$	$M_S \not\sqsubseteq M_D$ and $M_{KL} \not\sqsubseteq M_{\cancel{KL}}$
$(\sqsubseteq, \mathcal{D}_{KL}) \not\sqsupseteq\not\sqsubseteq (\sqsubseteq, \mathcal{S}_i)$	$M_S \not\sqsubseteq M_D$ and $M_{\cancel{i}} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{D}_{KL}) \not\sqsupseteq\not\sqsubseteq (\sqsubseteq, \mathcal{S})$	$M_S \not\sqsubseteq M_D$ and $M_{KL} \not\sqsubseteq M_{\cancel{KL}}$
$(\sqsubseteq, \mathcal{D}_{KL}) \not\sqsupseteq\not\sqsubseteq (\sqsubseteq, \mathcal{S}_{KLi})$	$M_S \not\sqsubseteq M_D$ and $M_{\cancel{i}} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{S}) \not\sqsupseteq\not\sqsubseteq (\sqsubseteq, \mathcal{S}_{KLi})$	$M_{KL} \not\sqsubseteq M_{\cancel{KL}}$ and $M_{\cancel{i}} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{D}) \not\sqsupseteq\not\sqsubseteq (\sqsubseteq, \mathcal{S}_{KLi})$	$M_S \not\sqsubseteq M_D$ and $M_{\cancel{i}} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{D}_{KLi}) \not\sqsupseteq\not\sqsubseteq (\sqsubseteq, \mathcal{S})$	$M_S \not\sqsubseteq M_D$ and $M_{KL} \not\sqsubseteq M_{\cancel{KL}}$

Figure 7. Incomparable slicing techniques.

$$
\begin{aligned}
M_S &= \{p''\}\\
&= \mathbb{M}_{p''}(\sqsubseteq, \mathcal{S}(V'', n''))\\
&= \mathbb{M}_{p''}(\sqsubseteq, \mathcal{S}_{KL}(V'', n''))\\
&= \mathbb{M}_{p''}(\sqsubseteq, \mathcal{S}_i(V'', {n''}^{k''}))\\
&= \mathbb{M}_{p''}(\sqsubseteq, \mathcal{S}_{KLi}(V'', {n''}^{k''}))
\end{aligned}
$$

$$
\begin{aligned}
M_D &= \{q''\}\\
&= \mathbb{M}_{p''}(\sqsubseteq, \mathcal{D}(\sigma'', V'', n''))\\
&= \mathbb{M}_{p''}(\sqsubseteq, \mathcal{D}_{KL}(\sigma'', V'', n''))\\
&= \mathbb{M}_{p''}(\sqsubseteq, \mathcal{D}_i(\sigma'', V'', {n''}^{k''}))\\
&= \mathbb{M}_{p''}(\sqsubseteq, \mathcal{D}_{KLi}(\sigma'', V'', {n''}^{k''}))
\end{aligned}
$$

The implication of these equations is similar to the above ones, namely $M_S \not\sqsubseteq M_D$, since $q''(\in M_D) \sqsubset p''(\in M_S)$.

Figure 7 shows how the above three counters examples are used to prove that the slicing techniques unconnected in the lattice in Figure 3 are not in relation according to the traditional syntactic ordering. □

Thus, Theorems 3 and 4 form an important result as they establish the connection between the two fundamental inter-technique relationships of subsumption and rank. The subsumption relationship tells us when one slicing technique can be used in place of another, while the rank relation tells us which produces the best (*i.e.*, smallest) slices. As a result of duality, the lattice of slicing techniques ordered by the traditional syntactic ordering is isomorphic to that for subsumption (in this case inverted), as shown in Figure 3.

6 Related Work

Program slicing was introduced by Mark Weiser in 1979 as a static program analysis and extraction technique [25]. In 1988 Korel and Laski [18] observed that slices would be more useful as a debugging aid, if they could be constructed dynamically, taking into account the execution characteristics which led to the observation of erroneous behavior. In 1990 Agrawal and Horgan [1] introduced four algorithms for constructing dynamic slices based on the Program Dependence Graph approach to slicing. However, the approach to dynamic slicing proposed by Agrawal and Horgan and that proposed by Korel and Laski differed as shown in our previous work [2].

Other theoretical work has attempted to lay the foundations of slicing. However, this previous work has been primarily concerned with static slicing and with intra-technique relationships rather than inter-technique relationships. For example, Reps and Yang [22] show that the PDG is adequate as a representation of program semantics, allowing it to be used in static slicing. Reps [20] shows how interprocedural-slicing can be formulated as a graph reachability problem, once again focusing on static slicing. Cartwright and Felleisen [6] show that the PDG semantics is a lazy semantics, because of the demand driven nature of the representation, while Giacobazzi and Mastroeni [12] present a transfinite semantics to attempt to capture the behaviour of static slicing. Harman et al. show the slicing is lazy in the presence of errors [16]. Weiser [25] observed that his slicing algorithm was not dataflow minimal and speculated on the question of whether dataflow minimal slices were computable. Danicic showed how this problem could be reformulated as a theorem about unfolding [8] while Laurence et al. [19] show how the problem can be expressed in terms of program schematology.

All this work has concerned static slicing. There has been very little formal theoretical analysis of the properties of dynamic slicing. The closest prior work to that in the present paper is the previous work of Venkatesh [24], who defined three orthogonal slicing dimensions, each of which offered a boolean choice. A slice could be static or dynamic, it could be constructed in a forward or backward direction and it could be either an executable program or merely a set of statements related to the slicing technique.

Venkatesh therefore considers 2^3 slicing techniques, some of which had not, at the time, been thought of before (for example the forward dynamic slice). As such, Venkatesh's work is was the first to attempt to consider inter-technique relationships. In particular, Venkatesh was concerned with the subsumes relationship. The present authors [2] also investigated the subsumes relationship using the projection theory of slicing. By contrast, the present paper is concerned with the rank relationship between slicing techniques.

There are several surveys of slicing: Tip [23], and Binkley and Gallagher [3] provide surveys of program slicing techniques and applications. De Lucia [10] presents a shorter, but more up-to-date survey of slicing paradigms. Binkley and Harman [4] present a survey of empirical results on program slicing. These papers provide a broad picture of slicing technology, tools, applications, definitions, and theory. Harman et al. [14, 13] introduced the projection theory used in this paper to analyse inter-technique slicing relationships. In this previous work the projection theory was used to explain the difference between syntax-preserving and amorphous slicing, whereas the present paper is concerned solely with syntax–preserving slicing.

7 Conclusion and Future Work

This paper has investigated the rank relationship between slicing techniques for static and dynamic slicing. Previous work has concerned the subsumes relationship. The primary contribution of the present paper is to show that the rank relationship is a mirror image of the subsumes relationship, leading to an inverted, but isomorphic lattice of inter-technique relationships. The paper also shows that the sets of minimal slices are useful in examining the relationships between slicing techniques.

Future work will extend this work to consider conditioned slicing, forward slicing and amorphous slicing. The ultimate goal of this research programme is to achieve a theoretical treatment of slicing that allows for formal reasoning about the relationships between slicing techniques in terms of important inter-technique relationships such as subsumption and rank.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, New York, June 1990.

[2] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and L. Ouarbya. Formalizing executable dynamic and forward slicing. In 4^{th} *International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, pages 43–52, Chicago, Illinois, USA, Sept. 2004. IEEE Computer Society Press, Los Alamitos, California, USA.

[3] D. W. Binkley and K. B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[4] D. W. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[5] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

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

[7] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.

[8] S. Danicic. *Dataflow Minimal Slicing*. PhD thesis, University of North London, UK, School of Informatics, Apr. 1999.

[9] S. Danicic, C. Fox, M. Harman, R. M. Hierons, J. Howroyd, and M. Laurence. Slicing algorithms are minimal for programs which can be expressed as linear, free, liberal schemas. *The computer Journal*. To appear.

[10] A. De Lucia. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[11] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, Aug. 1991.

[12] R. Giacobazzi and I. Mastroeni. Non–standard semantics for program slicing. *Higher-Order and Symbolic Computation*, 16(4):297–339, 2003. Special issue on Partial Evalution and Semantics-Based Program Manipulation.

[13] M. Harman, D. W. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.

[14] M. Harman and S. Danicic. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprehension (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[15] M. Harman, L. Hu, M. Munro, X. Zhang, D. W. Binkley, S. Danicic, M. Daoudi, and L. Ouarbya. Syntax-directed amorphous slicing. *Journal of Automated Software Engineering*, 11(1):27–61, Jan. 2004.

[16] M. Harman, D. Simpson, and S. Danicic. Slicing programs in the presence of errors. *Formal Aspects of Computing*, 8(4):490–497, 1996.

[17] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[18] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[19] M. R. Laurence, S. Danicic, M. Harman, R. Hierons, and J. Howroyd. Equivalence of conservative, free, linear program schemas is decidable. *Theoretical Computer Science*, 290:831–862, January 2003.

[20] T. Reps. Program analysis via graph reachability. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 701–726. Elsevier Science B. V., 1998.

[21] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *ACM Foundations of Software Engineering (FSE'94)*, pages 11–20, New Orleans, LA, Dec. 1994. ACM SIGSOFT Software Engineering Notes 19, 5 (December 1994).

[22] T. Reps and W. Yang. The semantics of program slicing. Technical Report Technical Report 777, University of Wisconsin, 1988.

[23] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[24] G. 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.

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

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