**H-0290 (H1008-010) August 16, 2010**
**Computer Science**

# IBM Research Report

## Program Sliding

**Ran Ettinger**
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

**Research Division**
**Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich**

**LIMITED DISTRIBUTION NOTICE**: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

# Program Sliding

Ran Ettinger
IBM Research – Haifa
rane@il.ibm.com

## ABSTRACT

As program slicing is a technique for computing a subprogram that preserves a subset of the original program's functionality, *program sliding* is a new technique for computing two such subprograms, a slice and its complement, the *co-slice*. A composition of the slice and co-slice in a sequence is expected to preserve the full functionality of the original code.

To avoid excessive code duplication, the co-slice generated by a sliding algorithm is designed to reuse the slice's results, correctly, in contrast to the re-computation performed by the complementary code generated by the best previous approach, called *tucking*.

A program sliding algorithm is presented along with its application in building refactoring tools. The ongoing construction of sliding-based refactoring tools for Java in the open-source project WALA, leveraging its Java slicer and integration into Eclipse, is reported.

## Keywords

Slicing, refactoring tools, reuse, software maintenance

## 1. INTRODUCTION

Program slicing, the study of meaningful subprograms that capture a subset of an existing program's behavior, can assist in building automatic tools for refactoring [6]. Slice extraction is the art of collecting a slice's set of not-necessarily contiguous program statements into a single code fragment, and reusing that fragment in the original code. With the goal of assisting programmers in maintaining high quality code, a solution to the problem of slice extraction along with its contribution to refactoring research are explored. The best previous solution, called tucking [13], suffers from low applicability, mainly due to high levels of code duplication. Such duplication results from the fact that some code is relevant not only for the extracted code but also for the remaining computation.

An advanced technique for the automation of slice extraction is introduced, through a family of highly non-trivial code motion transformations called *program sliding*. A sliding algorithm generates two subprograms, a slice and its complement, the *co-slice*, whose composition in a sequence preserves the original program's functionality.

For example, the code fragment on Figure 1(a), for reading some input numbers and printing their sum and product, can be replaced by a sequence of the two fragments on parts (b) and (c) of that figure. The slice, in part (b), is dedicated for the computation of the sum of the input numbers; while computing this sum, the slice has the side effect of populating the array of integers with elements read from the input stream; it also changes the value of a loop variable. The complementary code in part (c), in turn, correctly reuses the values in the array when computing their product; otherwise, it makes no further update to the input stream, and ignores the side effect on the loop index. As will be discussed in Section 8, the presented sliding algorithm is the first slice-extraction solution that supports correct isolation and reuse of the slice in such cases.

A high level description of a sliding algorithm is presented in Section 4. The details of how to compute the slice and co-slice are given in Section 6. The slicing and co-slicing algorithms are expressed in terms of a new program decomposition into non-contiguous entities called *slides*, formally introduced in Section 5.

To demonstrate the value of this new approach to the extraction of slices, Section 7 describes the application of sliding to previously documented yet unimplemented refactorings: Split Loop, Replace Temp with Query (RTwQ) and Separate Query from Modifier (SQfM). Sliding is also expected to facilitate the extraction of non-contiguous code in a general flavor of the well-known Extract Method refactoring. Such automation is crucial for enabling iterative and incremental software development [6]. It is also expected to impact on potential automation of bigger refactorings, as ambitious as Fowler and Beck's Separate Domain from Presentation or Convert Procedural Design to Objects [8].

The slicing, co-slicing, and sliding algorithms of this paper have all been proved correct for a simple imperative and sequential programming language; the language includes assignments, sequential composition of statements, conditionals, and loops. An ongoing attempt to adapt these algorithms to the real case of Java is hosted in the open-source T. J. Watson Libraries for Analysis project (WALA)[1], and benefits from the Java slicer in WALA and the integration to

*IBM Technical Report H-0290* August 2010, Haifa, Israel.
IBM Research – Haifa.

[1] **http://wala.sourceforge.net/**.

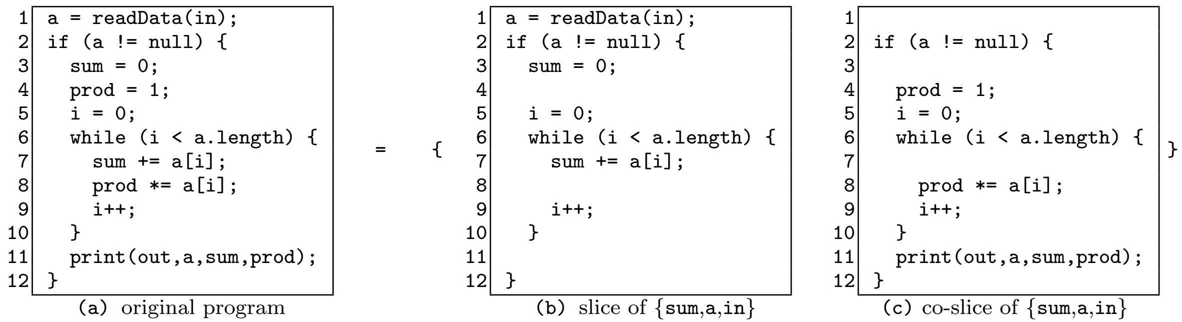


**Figure 1: A sliding example: the composition of the slice and its complement in a sequence yields a program functionally equivalent to the original; note the importance of (b) being before (c), for successful reuse of the extracted results.**

Eclipse. The prototype sliding implementation, along with prototype refactoring tools for RTwQ and SQfM, both reported in Section 7, are available to be downloaded from the WALA repository[2].

## 2. A SLIDING EXAMPLE

The key feature of sliding is its effective reuse, in the co-slice, of an extracted slice's results. In the example of Figure 1, the slice of variable `sum` is extracted, along with the array `a` and input stream `in`; the code of this extracted slice can be seen in part (b) of the figure. The extracted results act as input to the co-slice, as can be seen on lines 2,6,8 and 11 of part(c) of the figure. This way, there is no need to repeat the code for computing the extracted results, on lines 1,3 and 7. On the other hand, the other extracted statements, on lines 2,5,6 and 9, are duplicated as they are still needed in the co-slice.

Leaving the details of how this co-slice is computed for later in the paper (Section 6.2), note that had we managed to reuse the computation of `sum` but not that of `a`, in this example, we would have ended up recomputing `a` by repeating line 1 in the co-slice. And depending on the implementation of `readData()`, which is not included here, it is likely that a second invocation, in the co-slice, would have not returned the same result.

One way to avoid getting different results from such duplication is to make a backup of the initial value of some variables (in this case the input stream `in`) ahead of the slice, and restoring it ahead of the co-slice. The version of sliding presented in this paper, however, refrains from taking such measures. Instead, in cases where simple deletion of statements cannot be guaranteed to preserve the functionality of the original code, our sliding algorithm would announce failure, and a corresponding sliding tool would identify this situation and reject the transformation request. While a tool built on such a version of sliding is not the strongest possible, the refactoring algorithms of Section 7 show that this approach is powerful enough to assist in building advanced refactoring tools.

The two reasons for duplication, when composing a slice and its complement in a sequence, are control and data flow: some code such as the predicates in conditional and loop statements provide the control structure that may be relevant in both subprograms; and some code contributes to the computation of values that are relevant for the results of the two subprograms.

Sliding admits upfront the need to duplicate the mutual control structure, and focuses on keeping the need for recomputation low by reusing the results of the extracted slice in the co-slice. To that end, a new program decomposition is presented, of program *slides*, one that encapsulates the control structure required by each statement in a single entity. Figure 2 depicts the slides of all statements of the code fragment from Figure 1(a). With a given program being represented as the union of all its slides, the problem of slice extraction is reduced to the need to find a set of slides whose union is the required slice for extraction, and another set of slides whose union will yield a correct co-slice. In the example, the slice in Figure 1(b) is a union of the set of slides of lines $\{1, 3, 5, 7, 9\}$ while the co-slice of Figure 1(c) comprises the union of the set $\{4, 5, 8, 9, 11\}$.

## 3. PRELIMINARIES

The following background on program representation and several definitions regarding the scope of the program designated for extraction and its relevant state, in terms of the values of all its variables, will be needed for the precise description of a sliding algorithm.

### 3.1 Control Flow Graph

The control flow graph (CFG) of a code fragment is a labeled directed graph representing the order of execution of the individual statements of the program. The CFG of the code fragment in Figure 1(a) is shown on Figure 3.

A CFG has nodes $N$, typically with a single node $n \in N$ for representing each program statement and with two additional nodes, one for the *entry* the other for the *exit*; it has directed edges $E$ where each edge $(n_1, n_2) \in E$ represents the direct flow of control from its source $n_1$ to its target $n_2$; each node is the source of at most two edges: the exit node has no successors, normal nodes have one successor with no label on the connecting edge, and a predicate node corresponding to a conditional or a loop statement's condition has two successors, and each of the edges is labeled $T$ or $F$; however, those labels are irrelevant for slicing and will be insignificant in sliding too.

[2]See incubator project **com.ibm.wala.refactoring** in the developer information section of the WALA site.

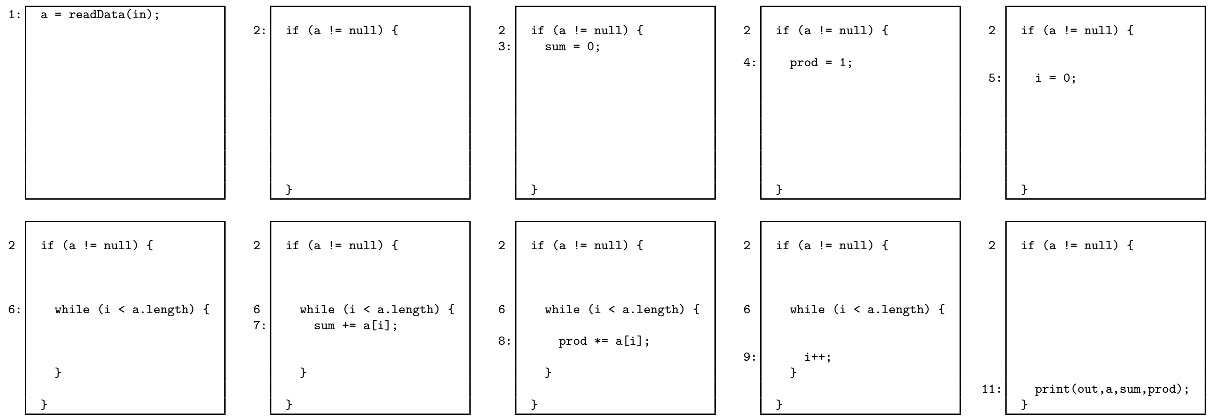


**Figure 2: The slides of the example code from Figure 1. Each slide represents a single statement (its line number is followed by a colon).**

## 3.2 Program Scope

Slice extraction, in this paper, is defined to work in the scope of a given fragment of code, say $S$, within the body of a program's procedure, say $P$. A solution to this slice-extraction problem will compute the slice of some given set of variables, say $V$, with respect to $S$. A transformed $P$, resulting from the replacement of $S$ with the sequence of the slice of $S$ on $V$ and its complement, should preserve the functionality of $P$.

For this transformation to be possible, it is common to expect the subgraph of the CFG of $P$ corresponding to $S$ to have a single entry node and a single exit node [13, 20]. This way, we can consider $S$ as represented by its own CFG, and forget about the enclosing code in $P$.

The fragment of lines 7-9 of Figure 1(a), represented by nodes 7-9 on the CFG of Figure 3, is an extractable fragment; so is the fragment of lines 3-5 and that of lines 3-10, with node 6 being its single exit. But the fragments of lines 3-8 or 7-11, where only a part of the loop's body is included, are not extractable; similarly, no fragment starting on line 1 or 2 is extractable unless it includes the full conditional, ending on line 12.

Considering syntax, or program structure, as slicing typically generates a subprogram of the original program by deleting irrelevant statements, let's refer by the term *sub-fragment* to the result of deleting some internal statements from a given code fragment.

Considering semantics, a slice is typically expected to preserve a subset of the original program's behavior. It is not, however, expected to preserve non-termination. That is, on input for which the original program does not terminate successfully, the slice is free to terminate [21]. Otherwise, slices would grow unnecessarily large, as any loop that does not modify any relevant variable will have to be included in the slice unless we can prove it definitely terminates. This does not mean that sliding will not be applicable for inherently non-terminating programs such as in reactive systems. Whenever the selected code fragment does not include the program's endless loop, but rather a portion of the loop's body, or a procedure called by that body, the slice extraction performed by sliding will be as relevant and applicable as in other imperative and sequential programs.

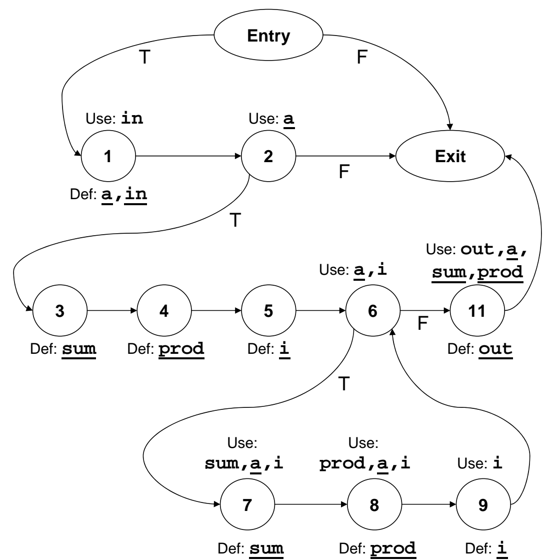


**Figure 3: A control flow graph (CFG) of the example code fragment of Figure 1(a). Final-use (Definition 12) and final-def (Definition 10) variable references are underlined.**

## 3.3 Program State and Termination

As stated above, we expect a solution to the problem of slice extraction to preserve the full functionality of the original code. In preserving functionality, we focus on the initial and final states of $S$, as represented by the value of all program variables. For this to work, any externally observable resource such as input and output devices should be represented internally as variables; accordingly, notice how the statement of node 1 in Figure 3 both uses the initial value and defines (*i.e.* assigns a new value to) the input stream variable, `in`; similarly, node 11 both uses and defines `out`. For simplicity, we assume no local variables are declared inside our fragments.

As shown on Figure 3, we refer to the set of variables each CFG node $n$ may modify as $Def(n)$ and to the set of

variables it refers to as *Use*($n$). Similarly, when considering the relation between the initial and final states of a code fragment $S$, we refer to the set of variables whose value on exit from $S$ might be different from their initial value as *Mod*($S$), and the set of variables whose initial value may impact the final state of $S$ as *Ref*($S$). Note that if a variable is always defined before being used in $S$, it should not be in its set of input variables, *Ref*($S$).

In the example, if $S$ denotes the entire code fragment of lines 1-12, we get $Mod(S) = \{\texttt{a}, \texttt{sum}, \texttt{prod}, \texttt{i}, \texttt{in}, \texttt{out}\}$ assuming `in` and `out` are modified on lines 1 and 11, respectively.

Since variables `a`,`sum`,`prod` and `i` are defined before ever being used, on the entire fragment, $S$, of lines 1-12 in Figure 1(a), we get $Ref(S) = \{\texttt{in}, \texttt{out}\}$, as the initial value of those two variables is used in lines 1 and 11, respectively. The input of the fragment 7-9 is $\{\texttt{a}, \texttt{i}, \texttt{sum}, \texttt{prod}\}$, and only `a` and `sum` are on the input of the extractable fragment 4-10.

An important concept of static program analysis relevant for computing slices is *reaching definitions*. A definition of a variable $v$ at a CFG node $n$ is reaching a CFG node $m$ iff there is a CFG directed path from $n$ to $m$ with the only definition of $v$ on $n$ itself. In our example, the definition of `sum` on node 3 reaches the entry to lines 4,5,6,7, and even 11 (as the loop might be executed 0 times), but not the entry to 8 or 9 because the assignment to `sum` on node 7 is the only one reaching those points; furthermore, this definition on line 3 does not reach the entry to node 1 or 2, as those nodes are not reachable at all from node 3, in the CFG.

More properties related to CFG paths, and the uses and definitions of variables, will be defined later, in Section 5, and used in Section 6 to compute slices and co-slices. The sets of modified and input variables, as well as the notion of reaching definitions, will be useful in the next section, for solving the problem of slice extraction.

# 4. A SLIDING ALGORITHM

Given a code fragment and a set of variables whose slice is to be extracted, we wish to isolate the computation of the final value of those variables from the other computations in the code fragment, such that both the extracted code and the complementary code (computing the other results) will consist of a version of the original code resulting from the deletion of some irrelevant statements. Ideally, the extracted code and the complementary code will have no code in common. However, as this will not always be possible, we should try to come up with the smallest possible complement, in order to avoid over duplication.

The success of reusing the final value of an extracted variable, in the co-slice, should not be taken for granted. It could be that the remaining computation requires the initial value of an extracted variable, instead of the final value, or perhaps even both; it might as well require some intermediate value of that variable. In any of those cases, no union of slides will yield the required program, since more variables, and some renaming of references will be required. In the example, had we been asked to extract the slice of `i`, we would not be able to reuse its final value, in the co-slice, inside the loop.

Given a code fragment $S$ and a set of variables $V$, the sliding algorithm in Figure 4 computes the extracted code $S_V$ and the complementary code $S_{CoV}$. It fails if correctness cannot be guaranteed, in this context.

The algorithm delegates the computation of the extracted code and the complement to the slicing and co-slicing algorithms of figures 7 and 8, respectively. It then concludes by checking that correctness can be guaranteed without a need for any backup of initial, intermediate, or final values. The condition to check that such backup variables are not needed is expressed in terms of the set of variables whose value might be modified in the slice, *Mod*($S$), and the set of variables whose initial value might affect the result of the co-slice, *Ref*($S_{CoV}$).

It is well known in slicing research that a minimal slice, one that includes the smallest number of statements, is not computable, in general [21]. (Knowing whether a definition of a variable in a loop may reach a use outside the loop requires knowledge that the loop may terminate.) Similarly, it is not possible, in general, to compute a minimal co-slice or minimal sets of modified and input variables. We therefore turn now to consider the semantic requirements from each of those before showing how to compute a safe approximation.

SLIDING($S, V$)
1 $S_V$ := COMPUTESLICE($S, V$)
2 $S_{CoV}$ := COMPUTECOSLICE($S, V$)
3 **assert** $V$ was reusable in its co-slice on $S$
(or the co-slicing algorithm would have failed)
4 **if** $(Mod(S_V) \cap Ref(S_{CoV})) \nsubseteq V$ **Fail.**
5 **return** $(S_V, S_{CoV})$

**Figure 4: A sliding algorithm generating a slice and co-slice with no need for backup variables. Its correctness depends on the slicing and co-slicing algorithms (of figures 7 and 8) satisfying the requirements of definitions 1 and 2, respectively.**

## 4.1 Requirements for Correctness

To ensure the correctness of the sliding algorithm from Figure 4, we investigate its impact on the final value computed for each variable, in the resulting program.

For correctness, the slicer and co-slicer must ensure their resulting code will terminate on any input that $S$ terminates on, and this code will have to compute the same results as those of $S$, for all variables, provided the refactored code is started on the same input. Our goal should be to have variables $V$ computed in $S_V$, leaving all other variables that act as input to $S_{CoV}$ unchanged; and then $S_{CoV}$ will need to complete the computation of all other variables $CoV$. Accordingly we define a slice and co-slice as follows:

DEFINITION 1 (SLICE). *A* slice *of a code fragment $S$ for a set of variables $V$ is a sub-fragment of $S$, say $S_V$, that when started in the same state as $S$, will terminate with the same final values in variables $V$ as $S$ will, provided $S$ terminates on the given initial state at all.*

Assuming variable $v$ will exit $S_V$ with the expected final value, the co-slicing algorithm will have to ensure it holds the same value on exit from $S_{CoV}$ too; we hence expect a co-slice of $S$ on $V$ not to modify the value of any variable $v \in V$.

Consider now a variable $cov \in CoV$, where $CoV$ is $Mod(S) \setminus V$. The final value of each such variable should be computed in $S_{CoV}$, with potential contributions to the computation flowing into $S_{CoV}$ through variables $V$ from $S_V$. As stated above, since no new variables are added to store initial values or intermediate results, variables $V$ may use only the

final value, as the slice $S_V$ would compute for them on the same input; again, as postulated above, on exit from $S_V$ all those variables will indeed hold the required values.

To ensure correctness of $S_{CoV}$, we demand that all other variables will hold their initial value on entry to $S_{CoV}$; this is ensured by the conditioned failure on Line 4, keeping any non-extracted variable whose value on entry to $S_{CoV}$ is relevant for correct execution, from being modified in the co-slice. This condition will prevent unintended and possibly incorrect dataflow from the slice to its complement.

Having the required initial state, we expect the correct final value of variable *cov* to be computed by the co-slice, $S_{CoV}$, in cases of termination.

DEFINITION 2 (CO-SLICE). *A* complementary slice *(or* co-slice *for short) of a code fragment S and a set of variables V, is a sub-fragment $S_{CoV}$ of S that computes the same results for all variables other than V if started in a state corresponding to the initial state of S, provided S terminates on that state, in the following way: all variables hold the same value as in the given initial state of S, except variables V; those variables hold the corresponding final value computed by S on that initial state.*

So we are left to present the slicing and co-slicing algorithms, and show that they yield sub-fragments that compute the same results as the original, on the set of sliced variables and on its complementary set, respectively.

# 5. PROGRAM REPRESENTATION

In this section we recall some definitions from the literature about the program dependence graph (PDG) and PDG-based slicing, and define a new program decomposition to serve as a basis for the construction of both the extracted slice and its complement. Our provably-correct slicing algorithm will yield the same results as PDG-based slicing.

## 5.1 Background on Program Dependence

DEFINITION 3 (POSTDOMINANCE). *A node n* postdominates *a node m in a program's CFG iff every path from m to the exit includes n.*

In the example, node 9 postdominates nodes 7 and 8 but not nodes 3-6, due to the CFG path from those to the exit without entering the loop body, bypassing it with the edge from node 6 to node 11.

DEFINITION 4 (CONTROL DEPENDENCE). *A CFG node n is* control dependent *on a CFG node m iff n postdominates a successor of m, but n does not postdominate m itself.*

Back in the example, node 9 is control dependent on node 6 because 9 postdominates 7 but not 6 itself. Note that node 9 is not control dependent on node 2, as 9 does not postdominate either successor of 2. Note also that each node that postdominates the $T$-successor of the entry is control dependent on the entry node, due to the special construction of the CFG's entry as a pseudo-predicate with the exit node as its $F$-successor.

DEFINITION 5 (DATA DEPENDENCE). *A CFG node n is* data dependent *on a CFG node m iff m defines a variable v that is used in n, and there is a path from m to n in the CFG with no further definitions of v.*

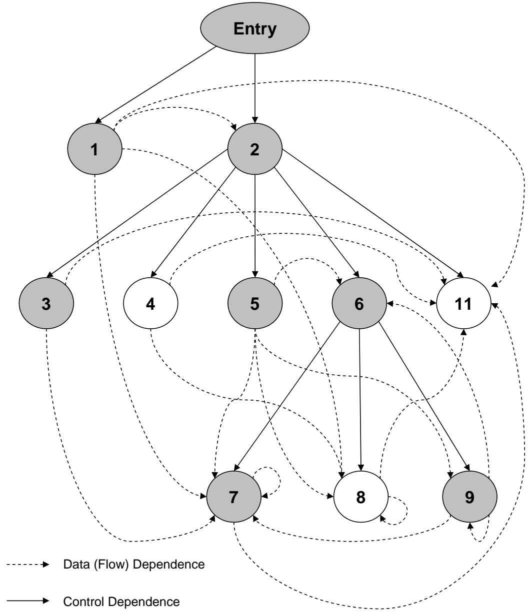


**Figure 5: A program dependence graph of the example code fragment of Figure 1(a). All nodes in the slice of 7 are highlighted.**

We will further stress that $n$ is data dependent on $m$ due to variable $v$. This will help us decide, later on, whether to consider a dependence when computing a co-slice.

DEFINITION 6 (PDG). *The* program dependence graph *(PDG) corresponding to a given program's CFG is a directed graph with the same nodes as in the CFG (except the CFG's exit node) and with an edge directed from node m to node n iff n is control or data dependent on m.*

The PDG of the example program is depicted on Figure 5. PDG-based slicing, when started with a set of nodes, say $C$, finds all nodes from which there is a directed PDG path to any node $c \in C$ [18]. A PDG-based slice staring from node 7, where `sum` is defined, consists of the nodes $\{Entry, 1, 2, 3, 5, 6, 7, 9\}$. One path causing the inclusion of node 1 goes through nodes 2 and 5, using a data dependence edge followed by two control dependence edges. The same resulting PDG-based slice would be computed for the set of nodes 1, 4 and 7, where all definitions of `a`, `in` and `sum` occur.

## 5.2 Slides and the Slide Dependence Graph

DEFINITION 7 (CONTROLLING NODES). *The set of controlling nodes of a PDG node n consists of the PDG control-dependence ancestors of n, i.e. the nodes from which there exists a directed path of control-dependence edges ending in n.*

The entry node, as well as nodes 2 and 6 are the controlling nodes of node 7 in the example PDG of Figure 5.

DEFINITION 8 (SLIDE). *A* slide *of a given code fragment S and one of its statements corresponding to a node n of the CFG of S, is the sub-fragment $S_n$ resulting from*

*deleting each statement of $S$ whose CFG node is neither $n$ itself nor is it a controlling node of $n$.*

All slides of the example program are shown on Figure 2. The slide of line 7, for example, is the sub-fragment resulting from the deletion of statements 1,3,4,5,6,9 and 11. This leaves us with the lines corresponding to node 7 and its controlling nodes 6 and 2. Note how in terms of concrete syntax, lines 10 and 12, for closing the blocks started on 6 and 2, respectively, were not deleted. In practice it is simpler to delete excluded statements than constructively combine all included statements, since the included statements are more likely to be non-contiguous. The preservation of concrete syntax, wherever possible, is particularly important in refactoring tools. Without it, there is a risk that programmers would not feel familiar with the refactored code and would be less inclined to using the tool. In this sense, slicing is suitable for the construction of refactoring tools.

Definition 9 (Union of Slides). *The union of a set of slides of a code fragment $S$ is the sub-fragment of $S$ resulting from deleting all statements whose node is not on any of the slides to be united.*

In this paper, a slide may interchangeably refer to the actual code sub-fragment it stands for, as defined above, or to the set of PDG/CFG nodes included in this sub-fragment, expecting that the intent will be evident from the context. Similarly, the slicing and co-slicing algorithms will collect and return a set slides, or equivalently the union of that set, or simply the program fragment corresponding to that union. The slides of a code fragment will be further categorized based on properties of the variables their nodes use or define, as follows:

Definition 10 (Final-Def Node). *Given a code fragment $S$, a CFG node $n$, and a variable $v$, we say that $n$ is a* final-def node *of $v$ in $S$ iff $v$ is defined in $n$ and there exists a path in the CFG from $n$ to the exit, free of definitions of $v$.*

It is common to say in such cases that the definition of $v$ at $n$ reaches the exit. Accordingly, note that each reaching definition of $v$ at the exit stand for a final-def slide of $v$.

Definition 11 (Final-Def Slide). *Given a code fragment $S$ and a slide $S_n$ corresponding to a CFG node $n$, the slide $S_n$ is a* final-def slide *of $S$ with respect to variable $v$ iff $n$ is a final-def node of $v$ in $S$.*

Definition 12 (Final-Use Node). *Given a code fragment $S$ and a variable $v$, a node $n$ on the CFG of $S$ is a* final-use node *for $v$ in $S$ iff $v$ is used in $n$ and each path in the CFG from $n$ to the exit is free of definitions of $v$.*

Definition 13 (Non-Final-Use Slide). *Given a code fragment $S$ and a slide $S_n$ corresponding to a CFG node $n$, the slide $S_n$ is a* final-use slide *of $S$ with respect to variable $v$ iff $v$ is not defined in $S_n$ and each CFG node $m \in S_n$ on which $v$ is used is a final-use node for $v$ in $S$.*

*Accordingly, $S_{n'}$ is a* non-final-use slide *of $S$ on $v$ iff $v$ is either defined on any CFG node $m$ of $S_n$ or it is used in $m$ and $m$ is not a final-use node of $S$ for $v$.*

Note that according to this definition, a slide may be neither a final-use slide nor a non-final-use slide of $S$ on $v$ if none of its nodes is using or defining $v$. The non-final-use slides of the set {`sum`, `a`, `in`} in the example, are slides 1,3 and 7. All uses of variable `a` on all the other slides are of its final value, and so is the use of `sum` on slide 11.

Definition 14 (Slide Dependence). *There is a* slide dependence *due to variable $v$ between the slides $S_m$ and $S_n$ of CFG nodes $m$ and $n$, respectively, iff there is a definition of $v$ in $m$ that reaches any node $n' \in S_n$ and $v$ is used in $n'$.*

Definition 15 (SlideDG). *The* slide dependence graph *(SlideDG) representing a code fragment $S$, in correspondence with the fragment's CFG, is a directed labeled graph with a node for each slide $S_n$ of $S$ corresponding to a CFG node $n$, and with an edge directed from a slide $S_m$ to a slide $S_n$ iff $S_n$ is slide dependent on $S_m$. An edge from $S_m$ to $S_n$ is labeled with the set of variables $U$ causing the dependence (i.e. there is a slide dependence from $S_m$ to $S_n$ due to variable $u$ iff $u \in U$).*

The SlideDG of the example code fragment is depicted on Figure 6. Note how no slide depends on slide 2 or 6, since their respective sets of defined variables are empty; those two slides are therefore excluded from the figure.

# 6. MEANINGFUL SETS OF SLIDES

## 6.1 Slide-Based Slicing

A slicing algorithm to fulfill the requirements of Definition 1 is presented in Figure 7. Given a code fragment $S$ and a set of variables $V$, the algorithm computes the set of slides whose union results in a valid slice $S_V$ of $S$ for the final value of variables $V$.

The algorithm starts by building the SlideDG and collecting the set $S_V$ of final-def slides of variables $V$, *e.g.* slides $\{S_1, S_3, S_7\}$ in the example, when $S$ is the code of Figure 1(a) and $V$ is {`sum`, `a`, `in`}. It then adds to $S_V$ (shown on the top row of Figure 6) all predecessor slides of members of $S_V$, iteratively. In the example, slides $S_5$ and $S_9$ are such predecessors, due to the initialization and increment of the loop variable, respectively.

Theorem 16. *A sub-fragment $S_V$ computed by the algorithm of Figure 7 for a code fragment $S$ and a set of variables $V$, is a correct slice of $S$ on $V$ as specified by Definition 1.*

Proof. For correctness, this algorithm is based on a similar slicing algorithm that has been proved to satisfy the requirements of computing the same results on all variables $V$ when $S$ is terminating [6], with one key difference. There, the slide of a variable was defined to include *all* assignments to that variable, along with their controlling statements. Accordingly, the slide dependences were defined in a flow-insensitive manner, rather than by using the CFG. So each referenced variable, on a slide, caused dependence on the slide of that variable, regardless on whether those assignments may flow into that reference. This construction made it relatively easy to prove correctness for the simple programming language used in that work, using the predicate calculus and program semantics of Dijkstra and Scholten [5]. Of course, such slices are unnecessarily large, especially when compared with the results of traditional,

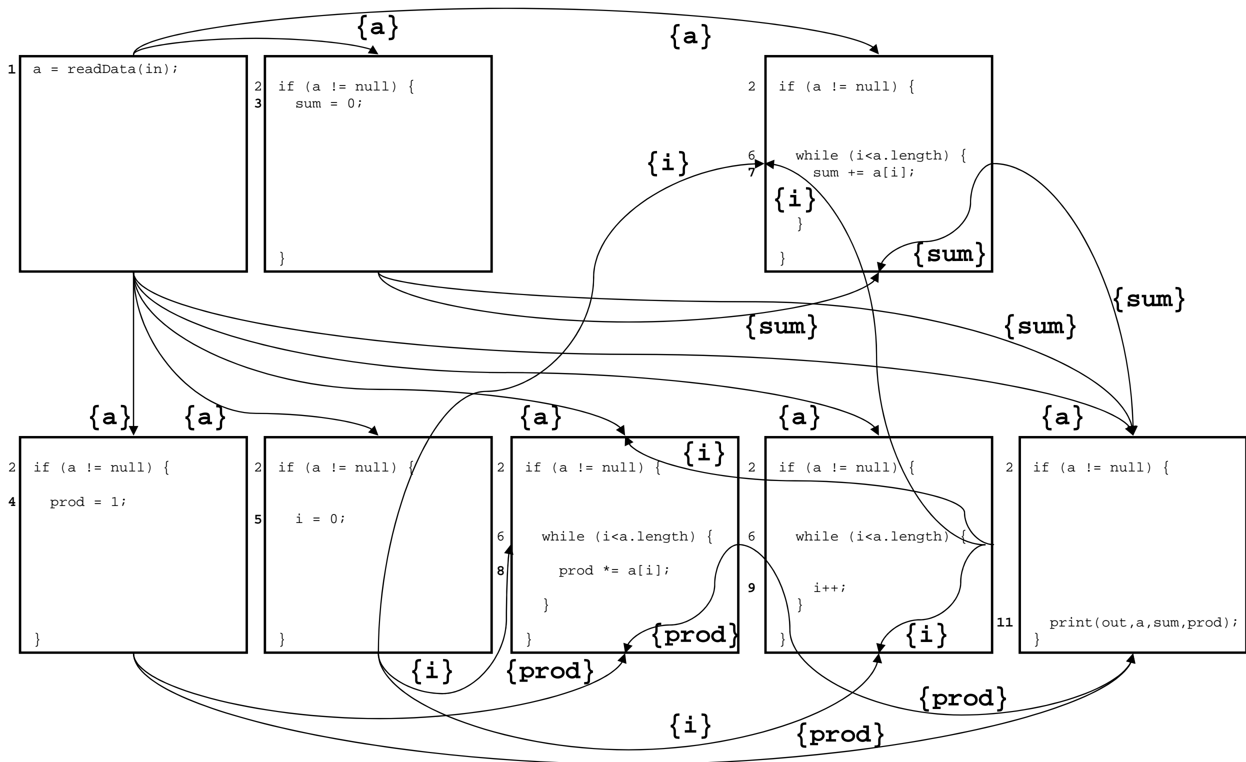


**Figure 6: A slide dependence graph (SlideDG) of the example code of Figure 1(a).**

flow-sensitive, slicing algorithms such as the original work by Weiser [21] and the popular PDG-based solution [18].

To gain flow-sensitivity, while maintaining correctness, the slides-based slicing algorithm of [6] was made to work on the static single assignment (SSA) form of the program [4]. This form is essentially a CFG with an extract feature, namely that each variable may not be defined in more than one node; so moving into SSA requires the splitting of a variables into a number of variable instances; and to avoid conflicts when more than one definition may reach a certain reference to a variable's value, new variable instances are added at static control-flow merge points, to contain the required value on each time this reference point is reached at execution time. The transformation of the original code into SSA, and back from it, after slicing, has been proved correct too. A key component of this proof shows that the merging of all instances of each variable, after slicing, when going back from SSA to the original form, using the original variable names, is indeed possible. Such merge of instances is not possible in general, after the SSA form has been further transformed, since two or more variable instances may be live simultaneously; if those instances cannot be shown to hold the same value in such points where both their values appear to be relevant, their merger might be incorrect.

The correctness of this original slides-based slicing algorithm using SSA will render our direct slides-based slicing algorithm of Figure 7 correct too, if we can only show the slices resulting from either algorithm are identical.

Suppose we transformed our code fragment to SSA and constructed our slides and slide dependence graph on that program representation. As stated above, the single assignment property ensures that the slide of each SSA node, say $n$, for a variable instance defined in $n$, say $v_i$, will be identical to the slide defined to include *all* assignments to $v_i$. Similarly, the single assignment property of the SSA form ensures that each variable reference incurs at most one slide dependence. (The exception is in any reference to a variable's initial value, where no slide dependence is incurred.) Therefore, again, the slide dependence as defined in this paper is identical to the slide dependence defined in the original, provably correct, SSA-based algorithm. So applying our algorithm on the SSA form will yield the same result as the original and this result has been shown to be amenable for successful return from SSA.

Accordingly, the correctness of our slicing algorithm will follow from showing that performing the algorithm directly on the CFG-based slides yields identical results to performing it on the SSA and then returning to the original form.

When using a general SSA form for slicing, it has been shown [2] that the resulting slices may be unnecessarily large. The problem mainly occurs in merge nodes of more than two predecessors; it is common in unstructured code, but may also occur in structured code, say when a conditional statement is followed by a loop: the loop entry has the end of the two sides of the conditional as predecessors as well as the end of the loop's body. When a variable $v$ is defined in any side of the conditional, but not in the loop, the merge of its three predecessor instances will be control dependent on the loop's predicate, causing at least a part of the loop to be added to any slice that requires the definition of $v$ in the conditional. Accordingly, in order to ensure the equivalence of results of slicing using CFG-based slides and SSA-based slide, we must

```
ComputeSlice(S, V)
1  (N, E) := SlideDG(S)
2  S_V := ⋃_{v∈V} {final-def slides of S on v}
3  Worklist := S_V
4  while Worklist ≠ ∅
5      S_n := Poll(Worklist)
6      forall S_m, U such that (S_m, S_n, U) ∈ E do
7          S_V := S_V ∪ {S_m}
8  return S_V
```

**Figure 7: A slicing algorithm.**

avoid this weakness of the SSA. When focusing our attention on correct slicing of programs of a simple programming language, with structured constructs only, such as loops and conditionals with a single exit, we avoid the problem mentioned above by insisting on two-way merge nodes, as in [6]. So the conditional followed by a loop will have one merge node for the conditional, flowing into a second merge node for the loop. This way, a conditional's merge node will not have a control dependence on the loop's predicate.

□

When the goal is to support the slicing, sliding, and refactoring of real-life programs, a practical approach would be to forgo the desire to formulate and construct a provably-correct solution, and replace it with some engineering techniques borrowed from the slicing literature. For example, for supporting dynamically allocated objects with potential aliasing, it is common to use and extend the program dependence graph, *e.g.* by adding new types of dependences or by redefining the existing definition of a data dependence. Our prototype implementation for Java is based on such a practical extension of the PDG. But back in the realm of a simple programming language, let's consider the similarity of our slides-based slicing with the popular PDG-based approach.

As mentioned above, the result of a traditional PDG-based slicing algorithm [18] is a collection of PDG/CFG nodes. Considering that each slide stands for a set of CFG nodes, and considering that a result $S_V$ of the slicing algorithm from Figure 7 is a set of slides, we can also think of $S_V$ as the union of all CFG nodes on its slides.

Theorem 17. *Let $S$ be a code fragment and $V$ a set of variables of interest. The set of CFG nodes corresponding to the PDG-based slice of all final-def nodes of $V$ in $S$ is identical to the set of CFG nodes on the slides-based slice, $S_V$, computed using the algorithm of Figure 7.*

Proof. If a node $m$ is in the PDG-based slice due to a PDG path $p$ to a final-def node $n$ of variable $v \in V$, it is also in the slides-based slice of $V$, by induction on the number of data dependence edges on $p$. When there is no data dependence edge on the path, $m$ either controls $n$ or it is $n$ itself; in such cases, $m$ is on the slide $S_n$ corresponding to node $n$, and therefore enters the slice $S_V$ on line 2 of the algorithm. When $p$ has $l > 0$ data dependence edges, we can split $p$ into three segments: a possibly empty path made exclusively of control dependence edges, from $m$ to a node $d$, a single-edge path from $d$ to a node $u$, due to a data dependence edge, and a path from $u$ to $n$. Clearly, $u$ is in the PDG-based slice, due to the third segment, and since it starts a path to $n$ with fewer than $l$ data dependence edges, the induction hypothesis ensures $u$ is in the slides-based slice $S_V$ too. By the definition of slide dependence, any slide $S_{u'}$ that includes node $u$ depends on the slide $S_d$ corresponding to node $d$. Accordingly, $S_d$ will be added to $S_V$ on line 7 of the algorithm when the slide dependence graph's predecessors of $S_{u'}$ are considered. Now similar to the base case above, node $m$ is on $S_d$ and hence in the slides-based slice as required.

Conversely, if a CFG node $m$ is on a slide $S_{m'} \in S_V$ added to the slides-based slice due to a path $q$ of slide dependence edges from $S'_m$ to the slide $S_n$ corresponding to a final-def slide of $v \in V$, we prove $m$ is also on the PDG-based slice of the final-def nodes of $V$ by induction on the length of the path $q$. When $S'_m$ is $S_n$ itself, the zero-length slide dependence path $q$ translates to a control-dependence path from $m$ to $n$, due to the definition of a slide and in particular of a final-def slide: there, $v$ is defined in the CFG node $n$, and each other node on $S_n$, including $m$, controls $n$. (Note that a side-effect definition of $v$ on a node $n'$, if permitted, may occur on a slide $S''_n$ of a different node $n''$; in such a case, while the slide $S_{n'}$ is a final-def slide of $v$, the slide $S_{n''}$ is not, unless $v$ is also defined in $n''$ itself. Otherwise, there could be an $m$ on a final-def slide $S_{n''}$ of $v$ with no control dependence path to $n'$.) For the induction step, assume the path $q$ is of length $l > 0$ and let's split $q$ to its first edge from $S_{m'}$ to a slide $S_{m''}$ corresponding to the CFG node $m''$, and the rest of the path $q'$. According to the definition of slide dependence, there is a variable $w \in \mathit{Def}(m')$ and there is a CFG node $u$ on $S_{m''}$ such that $w \in \mathit{Use}(u)$ and the definition of $w$ in $m'$ reaches $u$. The slide $S_{m''}$ is clearly in the slides-based slice, and hence all its nodes are in the PDG-based slice due to the induction hypothesis. In particular, the node $u$ on $S_{m''}$ is in the PDG-based slice due to some PDG-dependence path $p'$. There must also be a data dependence edge in the PDG from $m''$ to $u$, due to the fact that the definition of $w$ in the former reaches the latter. And, finally, by the definition of slides, and since $m \in S_{m'}$, there is a control-dependence path, $p''$, from $m$ to $m'$. Accordingly, there is a PDG path from $m$ to the final-def node $n$, *i.e.* the concatenation of the control path $q''$, the data dependence edge $(m'', u)$, and the path $q'$, and therefore $m$ is in the PDG-based slice. □

## 6.2 Slide-Based Co-Slicing

A co-slicing algorithm to fulfill the requirements of Definition 2 is presented in Figure 8. Given a code fragment $S$ and a set of variables $V$, the algorithm computes a set of slides whose union results in a valid co-slice $S_{CoV}$ of $S$ for variables $V$, with their expected final value available for reuse on entry.

Like in the slicing algorithm of Figure 7, we compute the co-slice by collecting a set of slides, called $S_{CoV}$ this time. We initialize this set with the final def slides of the variables of interest (line 4). Here, instead of the set $V$, we care about the computation of its complementary set $CoV$ (line 3). The main loop considers each predecessor $S_m$ of a slide $S_n$ from $S_{CoV}$. Such $S_m$ is added to the co-slice only if its dependence of $S_n$ is due to at least one variable outside of $V$ (line 11). Whenever we add slides to the co-slice (lines 4 and 13), we check that they are not violating the expected reuse by including a reference to an intermediate or initial value of any member of $V$.

In the example, when $V = \{\texttt{sum}, \texttt{a}, \texttt{in}\}$, we get $CoV =$

```
COMPUTECOSLICE(S, V)
1  (N, E) := SlideDG(S)
2  NonFU := ⋃_{v∈V} {non-final-use slides of S on v}
3  CoV := Mod(S) \ V
4  S_CoV := ⋃_{cov∈CoV} {final-def slides of S on cov}
5  if S_CoV ∩ NonFU ≠ ∅  Fail.
6  Worklist := S_CoV
7  while Worklist ≠ ∅ do
8      S_n := Poll(Worklist)
9      assert S_n ∉ NonFU
10     forall S_m, U such that (S_m, S_n, U) ∈ E ∧ S_m ∉ S_CoV do
11         if U ⊈ V
12             if S_m ∈ NonFU  Fail.
13             S_CoV := S_CoV ∪ {S_m}
14 assert S_CoV ∩ NonFU = ∅
15 return S_CoV
```

**Figure 8: A co-slicing algorithm.**

{i, prod, out}, and the corresponding final-def slides are $\{S_4, S_5, S_8, S_9, S_{11}\}$. The set *NonFU* of non-final-use slides of {sum, a, in} collected on line 2 of the algorithm includes $S_1$, for its reference to the initial value of in, $S_3$ for its definition of sum, and $S_7$, for both the definition and reference to intermediate values of sum. Since none of the slides in $NonFU = \{S_1, S_3, S_7\}$ is also in the initial $S_{CoV} = \{S_4, S_5, S_8, S_9, S_{11}\}$, a failure on line 5 of the algorithm is avoided.

Next, the main loop considers the dependences of slides from $S_{CoV}$ on other slides. There are such dependence edges from $S_1$ to all members of $S_{CoV}$ due to a and two more edges from $S_3$ and $S_7$ to $S_{11}$ due to sum. Since both a $\in V$ and sum $\in V$ the algorithm never reaches lines 12 and 13, and therefore terminates successfully, without adding any further slide. The resulting code on the union of slides $\{S_4, S_5, S_8, S_9, S_{11}\}$ can be seen in Figure 1(c).

Had we not included the input stream in in $V$, we would have ended up including it in $CoV$ and failing on line 5 of the algorithm. This failure would have been due to $S_1$ being both a final-def slide of in, added to $S_{CoV}$ on line 4, and a non-final-use slide of a, added to *NonFU* on line 2.

THEOREM 18. *A sub-fragment $S_{CoV}$ computed by the algorithm of Figure 8 for a code fragment $S$ and a set of variables $V$, is a correct co-slice of $S$ on $V$ as specified by Definition 2.*

To see why this algorithm is correct let's take a bit of a detour, in order to benefit from the correctness of our slicing algorithm and the similarity between the requirements of a co-slice and a slice. In the course of this detour, let's ignore the requirement to find a sub-fragment, focusing on the semantic requirement from a co-slice. The eventual result will be shown to be a sub-fragment identical to that of the co-slicing algorithm. In this sense, that detour will provide us with a second co-slicing algorithm, an indirect one. the equivalence of results will mean that the formal derivation of the second algorithm provide us with a proof of correctness not only for the second, but rather for both algorithms.

The advantages of the first algorithm, in turn, are twofold: (1) more elegantly, through union of slides of the original code, it makes it apparent that the result is a sub-fragment; and (2) more practically, it leads to a more efficient sliding implementation by avoiding the need to compute both a slice of the original code and another slice of a modified version of that code, with the associated need to re-compute the internal program representation.

The detour for deriving the second algorithm will require the following two operations:

DEFINITION 19 (NORMAL/FINAL-USE SUBSTITUTIONS). *A normal substitution of variables $U$ by variables $V$ in a code fragment $S$, denoted with the postfix operator $S[U \setminus V]$ (as in e.g. [16]) is the code fragment given by $S$ with all references and definitions of a variable $u \in U$ replaced with the corresponding $v \in V$. This operation is well defined iff $U$ and $V$ are same-length sequences of distinct variable names, and $V$ is a fresh set of names, none of which occurs in $S$.*

*A final-use substitution of $U$ by $V$ on $S$, denoted $S[\text{final-use } U \setminus V]$, is similar to normal substitution, with one difference: only final-use references to a $u \in U$ is replaced with the corresponding $v \in V$. Definitions and references to intermediate or initial values remain unchanged.*

And we are ready to prove Theorem 18, as follows:

PROOF. As input to a co-slice of $S$ on $V$, we have the initial value of all variables other than $V$ available, and the final value, as computed by $S$ on the given input, in $V$ itself. Consider a similar construction, in which we also have the initial value of variables $V$ available in backup variables $iV$:

$$(V := iV \; ; \; S)[live \; CoV]$$

$=$ { let $fV$ be a fresh set of variable names, disjoint from $Glob(S)$ and $iV$; note that we get $V = fV$ after $S$}

$$(fV := V \; ; \; V := iV \; ; \; S)[live \; CoV]$$

$=$ { let $S' := S[\textit{final-use } V \setminus fV]$ }

$$(fV := V \; ; \; V := iV \; ; \; S')[live \; CoV]$$

$\sqsubseteq$ { let $S'_{CoV} :=$ COMPUTESLICE$(S', CoV)$ }

$$(fV := V \; ; \; V := iV \; ; \; S'_{CoV})[live \; CoV]$$

$=$ { provided $V \cap Glob(S'_{CoV}) = \emptyset$ }

$$(fV := V \; ; \; S'_{CoV})[live \; CoV]$$

$=$ { let $S_{CoV} := S'_{CoV}[fV \setminus V]$; the live variables clause is now redundant, since $Mod(S_{CoV}) \subseteq CoV$}

$$S_{CoV} \quad .$$

To summarize, the co-slice $S_{CoV}$ resulting from the derivation above is (COMPUTESLICE$(S[\textit{final-use } V \setminus fV], CoV))[fV \setminus V]$ and it is well-defined iff $V \cap Glob($COMPUTESLICE$(S[\textit{final-use } V \setminus fV], CoV)) = \emptyset$.

We now show that the two co-slicing approaches, of Figure 8 and of the derivation above are well-defined under the same conditions, and yield identical results. That is, for any given code fragment $S$ and a set of variables $V$, the code on COMPUTECOSLICE$(S, V)$ is identical to that of (COMPUTESLICE$(S[\textit{final-use } V \setminus fV], CoV))[fV \setminus V]$ and the former is successful to generate a co-slice iff the latter is well-defined.

In terms of slides, and the slide dependence graph, comparing the representation of $S$ with that of $S' := S[\textit{final-use } V \setminus fV]$, we observe that the only changes made by final-use substitution are in removing final-use dependences on the slides of $V$. (No other dependence comes in their place since $fV$ are not defined in $S'$.) Let $E_{fV}$ be the set of removed edges.

We prove by induction on the steps of the algorithm that the two co-slicing algorithms are successful under the same conditions, and yield identical results.

Since no definition has been modified by final-use substitution, the final-def slides of $CoV$ on $S$ and $S'$ are the same slides. Hence, the initialization of the result set of slides performed by both algorithms is identical. Slicing $S'$ for $CoV$ starts by collecting the final-def slides of $CoV$ on line 2 of Figure 7, calling it $S_V$, and the co-slicing algorithm on $S$ for $V$ does the same on lines 3-4 of Figure 8, into the set called $S_{CoV}$).

The co-slicing algorithm fails on line 5 if any final-def slide of $CoV$ either defines or makes a non-final reference to a variable $v \in V$. In such a case, the corresponding slide of $S'$ will be in the detour co-slice, and since only final uses are substituted, it will definitely define or make reference to $v$ and therefore be unsuitable for the subsequent step of normal substitution. On the other hand, if the test on line 5 passes, there is no such $v \in V$ on any final-def slide of $V$ and so the only references to $V$ are to final values, and those were substituted with references to $fV$, so the initial $S_V$ is definitely clear of definition of reference to $V$, as required.

Next, in the main loop, the slicing on $S'$ for $CoV$ adds the predecessor $S_m$ of a slide $S_n \in S_{CoV}$ if the set $U$ on the label of edge $(S_m, S_n, U)$ is not of a subset of $V$. If it is a subset of $V$, each reference in $S_n$ to a $u \in U$ must be to a final value, and we have $(S_m, S_n, U) \in E_{fV}$; so the detour algorithm will similarly not add $S_m$ in such a case. Now before adding $S_m$ to the co-slice, the first algorithm checks it is not in $NonFU$; if it is, the algorithm fails just like the second algorithm would fail in the final step of normal substitution (as explained in the initialization step).

So both algorithms fail under the same conditions and end up with the same sets of slides; but the second algorithm's result of the slicing step has final-use references to $V$ replaced with $fV$; the final step substitutes all those references back to the original name, making the two resulting co-slices literally identical.

□

Accordingly, our co-slicing algorithm of Figure 8, when successful on $S$ and $V$, yields a valid sub-fragment that computes the same results as the original program for all variables in the complementary set of variables $CoV$ (being $Mod(S) \setminus V$).

# 7. EVALUATION

To evaluate the usefulness of sliding, we have implemented a number of sliding algorithms in a tool integrated into the Java development environment in Eclipse, and applied it in the construction of two refactoring techniques.

## 7.1 Implementation

Inside the Java source-code editor in Eclipse, the user can select a variable of interest and add it to an accumulated list $V$; as the relevant fragment $S$, the current prototype tool focuses on the whole method's body. If one wishes to perform sliding on a smaller fragment, it should be possible to extract that fragment into its own method object, call it from the original, and collect the relevant final values from the extracted object. After performing sliding on that new method, the resulting code can be inlined back into its original method.

For slicing we use the WALA slicer, and for co-slicing, the current implementation performs the indirect algorithm described in Section 6.2. Compared with the algorithm of Figure 8, which has been proved to yield identical co-slices on a simple language, the implemented algorithm needs to compute slicing after slightly changing the original program. This requires expensive re-computation of the WALA intermediate representation; we anticipate that a re-implementation of co-slicing based on the algorithm of Figure 8 will therefore perform better in terms of time and space. It is left to experimentation to verify that it indeed yields the same results in practice.

The current sliding tool includes an implementation of the algorithm of Figure 4 (with slicing and co-slicing computed differently, as stated above), and a number of more advanced algorithms, where additional variables are added to store the initial value of variables ahead of the slice and retrieve it ahead of the co-slice, and similar backup variables for storing the extracted results after the slice and retrieving them at the end, after the co-slice. Another advanced feature is to specify whether an extracted value should be reused of not; this way, one can avoid rejection when the successful reuse is not possible.

An additional sliding algorithm finds all variables whose slice is included in the extracted slice, and automatically attempt to reuse those too. This way, the user can ask to extract the computation of `sum`, in our example, and the tool will find that `a` should be extracted and reused too, and that `in` must be extracted too, to avoid rejection due to the duplication of line 1.

## 7.2 Sliding-Based Refactoring Tools

A direct application of sliding is the Split Loop refactoring [1]. We have yet to add a designated tool for splitting loops, but the user can apply it using the sliding tool. Extract the loop and the relevant initialization into a new method object, select the variables of interest, and apply sliding. If the initialization of the loop index is not in the selected code fragment (as in the fragment of lines 6-10 in our example) the tool based on Figure 4 would correctly reject the transformation, as the loop in the co-slice would be skipped. One of the more advanced versions described above would add a backup variable for the initial value of that loop index.

For the refactoring Replace Temp with Query (RTwQ) [8] we have implemented a tool with similar user interface as that of Extract Method. This refactoring involves the extraction of the computation of a single variable, a temporary one, into a method of its own. That method may have no side effects and it will be invoked from all places in the original code where the final value of the temp was used. Invoking RTwQ on the variable `prod` of the co-slice on Figure 1(c), assuming this variable was local, would replace the use of `prod` in line 11 with a call to the new method, and the loop would be eliminated from the original code (assuming `i` was local too). Invoking it on the original program of Figure 1(a) would be rejected, correctly, since the extracted slice would involve side-effects on the input stream.

For invoking the tool, the user selects the local variable of interest, instead of a fragment of code in Extract Method, and the tool prompts the user to specify a name for a new method. The tool performs the transformation in three steps: sliding, followed by Extract Method on the extracted

slice and by Inline Temp on the selected variable [6]. The sliding is given the selected local variable as $V$ and extracts its slice; the sliding may fail, and correctly so, if a non-final value of the temp is still relevant in the co-slice; moreover, since the query may have no side effects, the tool must fail if any global variable is modified in the slice.

An additional refactoring we have implemented is for Separate Query from Modifier (SQfM) [8]. This refactoring involves the splitting of a non-void method with side effects to two methods. Like in RTwQ, the extracted slice is of a single variable, the one holding the returned value from the original method. (If there is no such temp, we add it first.) After sliding the slice of this temp away from the computation of the modifier, *i.e.* the code with the side effects, we perform Extract Method twice, on the slice and co-slice, and finally perform Inline Method, to replace all calls to the original method with the two invocations, of the query and modifier. The SQfM tool can be useful also for temporarily updating the code, unlike in a typical refactoring scenario, for testing, debugging, or verification, *e.g.* by enabling the application of tools and techniques that assume no side effects exist in conditional expressions.

# 8. RELATED WORK

Earlier research on extracting slices from existing systems, in the context of software reverse engineering and reengineering, has focused mainly on how to discover reasonable slicing criteria [3, 14]. In the context of refactoring tools, it is common to leave the choice of what to extract to the programmer.

The earliest mention of an interactive process for behavior-preserving method extraction [17, 9] considered the extraction of contiguous code only.

Maruyama [15] proposed a scenario for the extraction of the slice of a single variable from a selected fragment, or block of statements, into a new method; a call to that method is placed ahead of the code for the remaining computation. The importance of proving correctness is mentioned, but no proof is given. The suggestion to base the proof on a theorem that two programs are equivalent if they have isomorphic PDGs [10] would prohibit the desired duplication of nodes. A more recent work building on that approach [19] suffers from the same limitation.

A number of provably correct algorithms for the extraction of a set of not-necessarily contiguous statements have been proposed in the literature [13, 11, 12].

Of those, tucking [13] is most generally applicable for isolating the slice of a code fragment. Tucking starts by adding to the statements designated for extraction all other statements in their slice, limited to the smallest fragment enclosing those statements. If we apply this algorithm by selecting such a slice in the first place, no other statement would be added to the extracted code. This is unfortunately not the case in the algorithm of Komondoor and Horwitz [12], where each statement that the algorithm is unable to move away from the slice, correctly, is added to the extracted code. In the worst case, this approach extracts the whole fragment, essentially leaving it unchanged. In particular, no assignment can be duplicated and loop statements can either be extracted fully, or not extracted at all. Therefore, splitting loop as in our example of sum and prod, is not possible by that algorithm. Komondoor and Horwitz had an earlier algorithm [11] in which all permutations of the selected statements were considered, in looking for an arrangement of statements in which all selected statements are contiguous and where all control and data dependences are preserved. This algorithm does not permit any duplication, not even of conditionals, and may therefore be applicable for slice extraction only in cases where each predicate in the slice appears in it along with all the statements it controls.

So tucking is the only previous solution to slice extraction that can untangle a loop that computes more than one result, as in the example of sum and prod [7]. In tucking, however, the complementary code is computed as the slice from all non-extracted statements, so no reuse of the extracted results is possible. In our example, that complement would include the whole fragment of lines 1-12, as it would start slicing from nodes 4,8 and 11 of figure 5. The duplication of the computation of `sum`, in this case, is undesirable but still correct. The problem is with the duplication of node 1, for reading integers from the input stream: tucking's precondition states that a variable whose value on exit from the fragment is relevant, may not be defined in both the slice and its complement; in this case, even if assuming the values of `i`,`sum` and `a` are not relevant after this fragment, the input stream `in` clearly is, and the transformation would have to be rejected.

The idea of allowing data to flow from the extracted code to the complement, in sliding, is based on the two Komondoor and Horwitz algorithms [11, 12].

# 9. CONCLUSION

To paraphrase Weiser's seminal work [21], sliding is a new way of recomposing programs automatically. Limited to code already written, it may prove useful during the refactoring, testing, and maintenance portions of the software life cycle. This paper concentrated on the basic methods for sliding programs and their embodiment in automatic tools for refactoring. Future work on sliding-based programming aids is necessary before the implications of this kind of recomposition are fully known.

## Acknowledgements

I gratefully acknowledge the writing guidance of Cindy Eisner and the implementation work of Alex Libov, Eli Kfir, Daniel Lemel, Dima Rabkin, and Vlad Shumlin.

# 10. REFERENCES

[1] An online refactoring catalog. http://www.refactoring.com/catalog/.

[2] A. Abadi, R. Ettinger, and Y. A. Feldman. Improving slice accuracy by compression of data and control flow paths. In *7th Joint Mtg. European Software Engineering Conf. (ESEC) and ACM Symp. Foundations of Software Engineering (FSE)*, Aug. 2009.

[3] A. Cimitile, A. D. Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: a case study. In *ICSM*, pages 124–133, 1995.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[5] E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[6] R. Ettinger. *Refactoring via Program Slicing and Sliding*. PhD thesis, University of Oxford, Oxford, United Kingdom, 2006.

[7] R. Ettinger and M. Verbaere. Untangling: a slice extraction refactoring. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 93–101, New York, NY, USA, 2004. ACM Press.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 2000.

[9] W. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering*, 2(3):228–269, July 1993.

[10] S. Horwitz, J. Prins, and T. W. Reps. Integrating non-interfering versions of programs. In *POPL*, pages 133–145, 1988.

[11] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–169, New York, NY, USA, 2000. ACM Press.

[12] R. Komondoor and S. Horwitz. Effective automatic procedure extraction. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, 2003.

[13] A. Lakhotia and J.-C. Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology*, 40(11-12):677–690, 1998.

[14] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. Software Eng.*, 23(4):246–259, 1997.

[15] K. Maruyama. Automated method-extraction refactoring by using block-based slicing. pages 31–40. ACM Press, 2001.

[16] C. Morgan. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[17] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[18] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[19] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities. In *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 119–128, Washington, DC, USA, 2009. IEEE Computer Society.

[20] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *ICSE*, pages 172–181, 2006.

[21] M. Weiser. Program slicing. In *ICSE*, pages 439–449, 1981.