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.

The co-slice generated by sliding is designed to reuse the slice's results, correctly, in order to avoid re-computation causing excessive code duplication. By isolating coherent slices of code, making them extractable and reusable, sliding is shown to be an effective step in performing advanced code refactorings.

A practical sliding algorithm, based on the program dependence graph representation, is presented and evaluated through a manual sliding-based refactoring experiment on real Java code.

Keywords: Program slicing, sliding, co-slicing, reuse, refactoring.

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 [4]. 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.

An advanced technique for the automation of slice extraction is introduced, through a family of code motion transformations called *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. When preservation of functionality cannot be guaranteed, a sliding tool would warn the user and offer corrective measures, known in the literature as *compensation*, or compensatory code.

Deviating from earlier practices of code extraction, where the input to the transformation includes some selection of statements to be made contiguous, sliding takes a set of variable names V as input, expecting to turn the slice of code for computing the final value of V into a contiguous fragment; and while different sets of variables $V1$ and $V2$ may have the exact same slice, the co-slice computed by sliding to each set may differ. For example, in the `putInCacheIfAbsent` method shown on Fig. 1, taken from the Java compiler in Eclipse, the set of statements {1,2,6,8,9,11,13-19,21,22} (see top part of Fig. 2) forms the slice of V = {`index`}, in the scope of that method, as well as the slice of V^+ = {`index,k2V,entry,cAC,cAC.*,k2V.*`}, or the slice of any set being both a subset of V^+ and a superset of V. (Note the use of acronym of camel-case variable names, for brevity. This set V^+ will be used throughout the paper.) After sliding for V, Fig. 3 shows the result of `index` from the slice is used in the co-slice on line 23. The co-slice generated by sliding of the larger set V^+, shown in the bottom part of Fig. 2, reuses more results of the slice, such as `cAC` on line 20 and `entry`, defined on line 14 of the slice and used on line 15 of the co-slice. This reuse of local variables is enabled by moving their declaration to an outer scope. One benefit of this reuse is the reduced level of code duplication: statements 8,14,17-19,21-22 were duplicated by the sliding of V but not by that of V^+. Another advantage of the further reuse and reduced duplication is the potential reduction in need for compensation. One disadvantage, on the other hand, of this extra reuse, is the need to move local declarations, potentially requiring the renaming of those whose original name is used for other variables in the extended scope. A more significant disadvantage of the reuse of local results is its impact on the ability to reuse the slice in other parts of the program. For this to work, we need to extract that slice as a reusable method. Since Java allows only one local value to be returned from a method, some alternative compensation would be needed, making the resulting code less attractive for some applications. Sliding provides the flexibility to choose between high levels of local reuse of multiple slice results and a more straightforward global reuse of a single slice result.

A sliding algorithm, along with the details of how to compute the co-slice, is presented in Sect. 3. To demonstrate the value of this new approach to the extraction of slices, Sect. 4 describes the application of sliding to previously documented 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 [4]. 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 [7].

The initial work on sliding was a theoretical pursuit. The original transformation rules have been proved correct for a simple imperative programming language restricted to assignments to primitive variables of cloneable types, sequential composition of statements, conditionals, and loops [4]. The current work presents a first practical sliding algorithm based on the program dependence graph (PDG) [6,9]. The results of a preliminary evaluation are reported in Sect. 5, and the relation to previous work is discussed in Sect. 6.

```
/**
 * @param key1 the given declaring class name
 * @param key2 the given field name or method selector
 * @param key3 the given signature
 * @param value the new index
 * @return the given index
 */
private int putInCacheIfAbsent(final char[] key1, final char[] key2,
                               final char[] key3, int value) {
    int index;
    HashtableOfObject key1Value = (HashtableOfObject)this.methodsAndFieldsCache.get(key1);
    if (key1Value == null) {
        key1Value = new HashtableOfObject();
        this.methodsAndFieldsCache.put(key1, key1Value);
        CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
        index = -value;
        key1Value.put(key2, cachedIndexEntry);
    } else {
        Object key2Value = key1Value.get(key2);
        if (key2Value == null) {
            CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
            index = -value;
            key1Value.put(key2, cachedIndexEntry);
        } else if (key2Value instanceof CachedIndexEntry) {
            // adding a second entry
            CachedIndexEntry entry = (CachedIndexEntry) key2Value;
            if (CharOperation.equals(key3, entry.signature)) {
                index = entry.index;
            } else {
                CharArrayCache charArrayCache = new CharArrayCache();
                charArrayCache.putIfAbsent(entry.signature, entry.index);
                index = charArrayCache.putIfAbsent(key3, value);
                key1Value.put(key2, charArrayCache);
            }
        } else {
            CharArrayCache charArrayCache = (CharArrayCache) key2Value;
            index = charArrayCache.putIfAbsent(key3, value);
        }
    }
    return index;
}
```

Fig. 1. Example code, ahead of sliding, taken from the Eclipse Java compiler's org.eclipse.jdt.internal.compiler.codegen.ConstantPool class

The main contributions of this paper are as follows:

- Practical co-slicing and sliding algorithms suitable for the real case of (sequential) Java, building on traditional (backward, static, syntax preserving) slicing and the underlying program dependence graph representation, hence deferring the responsibility for correctness, scalability, and applicability for more languages to the slicer and the dependence graph construction mechanism.
- First evidence for the applicability of sliding for solving known refactoring techniques, including a detailed account of how developers can use sliding as a building block for performing such refactorings.
- A preliminary evaluation, having transformed a well-tested massively-used real-life Java code with no detected regression.

```
       int index;
 1:    HashtableOfObject key1Value = (HashtableOfObject)this.methodsAndFieldsCache.get(key1);
 2:    if (key1Value == null) {
 6:        index = -value;
       } else {
 8:        Object key2Value = key1Value.get(key2);
 9:        if (key2Value == null) {
11:            index = -value;
13:        } else if (key2Value instanceof CachedIndexEntry) {
               // adding a second entry
14:            ~~CachedIndexEntry~~ entry = (CachedIndexEntry) key2Value;
15:            if (CharOperation.equals(key3, entry.signature)) {
16:                index = entry.index;
               } else {
17:                ~~CharArrayCache~~ charArrayCache = new CharArrayCache();
18:                charArrayCache.putIfAbsent(entry.signature, entry.index);
19:                index = charArrayCache.putIfAbsent(key3, value);
               }
           } else {
21:            ~~CharArrayCache~~ charArrayCache = (CharArrayCache) key2Value;
22:            index = charArrayCache.putIfAbsent(key3, value);
           }
       }
 1:    ~~HashtableOfObject~~ key1Value = (HashtableOfObject)this.methodsAndFieldsCache.get(key1);
 2:    if (key1Value == null) {
 3:        key1Value = new HashtableOfObject();
 4:        this.methodsAndFieldsCache.put(key1, key1Value);
 5:        CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
 7:        key1Value.put(key2, cachedIndexEntry);
       } else {
 9:        if (key2Value == null) {
10:            CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
12:            key1Value.put(key2, cachedIndexEntry);
13:        } else if (key2Value instanceof CachedIndexEntry) {
               // adding a second entry
15:            if (CharOperation.equals(key3, entry.signature)) {
               } else {
20:                key1Value.put(key2, charArrayCache);
               }
       }
23:    return index;
```

Fig. 2. A sliding example: the slice of $V^+ = \{index, k2V, entry, cAC, cAC.*, k2V.*\}$, followed by its complement, the co-slice

2 Preliminaries

The following background on program analysis and definitions regarding the scope of the program designated for extraction and its relevant state, will be needed for the precise description of a sliding algorithm.

2.1 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 in Fig. 1 is shown on Fig. 4. It is common to make the CFG compact by grouping nodes into basic blocks [2]. The granularity of individual statement nodes, however, is convenient for construction of the program dependence graph (PDG), as it is for slicing and sliding.

```
 1:   ~~HashtableOfObject~~ key1Value = (HashtableOfObject)this.methodsAndFieldsCache.get(key1);
 2:   if (key1Value == null) {
 3:       key1Value = new HashtableOfObject();
 4:       this.methodsAndFieldsCache.put(key1, key1Value);
 5:       CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
 7:       key1Value.put(key2, cachedIndexEntry);
      } else {
 8:       Object key2Value = key1Value.get(key2);
 9:       if (key2Value == null) {
10:           CachedIndexEntry cachedIndexEntry = new CachedIndexEntry(key3, value);
12:           key1Value.put(key2, cachedIndexEntry);
13:       } else if (key2Value instanceof CachedIndexEntry) {
              // adding a second entry
14:           CachedIndexEntry entry = (CachedIndexEntry) key2Value;
15:           if (CharOperation.equals(key3, entry.signature)) {
              } else {
17:               CharArrayCache charArrayCache = new CharArrayCache();
18:               charArrayCache.putIfAbsent(entry.signature, entry.index);
19:               ~~index =~~ charArrayCache.putIfAbsent(key3, value);
20:               key1Value.put(key2, charArrayCache);
              }
          } else {
21:           CharArrayCache charArrayCache = (CharArrayCache) key2Value;
22:           ~~index =~~ charArrayCache.putIfAbsent(key3, value);
          }
      }
23:   return index;
```

Fig. 3. An alternative co-slice for the same slice from Fig. 2. With less reuse of local variables, it duplicates more statements (8,14,17-19,21,22). Yet it is more appropriate for some applications, as the slice can be extracted into a new method, avoiding rejection due to "ambiguous results".

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 $(m, n) \in E$ represents the direct flow of control from its source m to its target n; each node is the source of at most two edges (with `switch` statements represented as nested `if`s): 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.

2.2 Program Scope and State

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 method, say M. 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 M, resulting from the replacement of S with the sequence of the slice of S on V and its complement, should preserve the functionality of M.

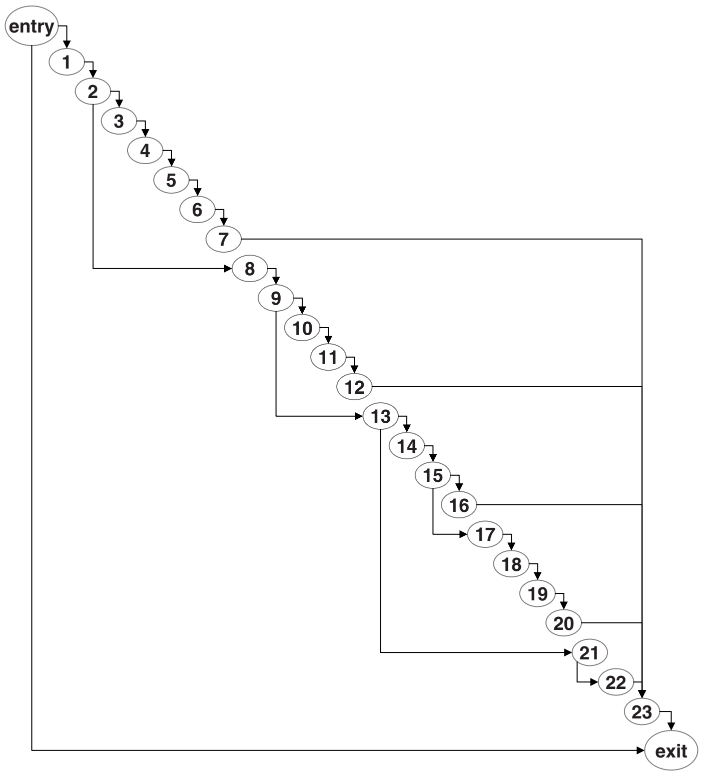


Fig. 4. A control flow graph (CFG) representation of the example code from Fig. 1. True or False labels on edges leaving predicate nodes are omitted as they are irrelevant for slicing and sliding. The entry is made into a pseudo predicate with the exit node as its other successor for convenience, making it the root of the control-dependence subgraph of the PDG.

For this transformation to be possible, the subgraph of the CFG of M corresponding to S is expected to have a single entry node and a single exit node [12,18]. This way, we can consider S as represented by its own CFG, and forget about the enclosing code in M. For such a single-entry-single-exit (SESE) region of the CFG to be extractable, a further requirement is that there is no edge from the exit node back to any node of the region.

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.

The set of variables each CFG node n may modify is denoted by $Def(n)$, and the set of variables it refers to is $Use(n)$. The returned value of a non-void method is given a name too, `<retval>`, and will be included in the set of defined variables whenever a `return` statement is in scope.

2.3 Background on Program Dependence

Definition 1 (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 20 postdominates nodes 17-20 but not node 15, due to the CFG path $< 15, 16, 23, exit >$. Node 23 postdominates all nodes except the entry.

Definition 2 (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 20 is control dependent on node 15 because 20 is a postdominator of 17 but not of 15 itself. Note that node 20 is not control dependent on node 13, as 20 does not postdominate either successor of 13. Note also that each node that postdominates the normal 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 other successor.

In terms of value transfer through program variables and objects, a variety of data dependence definitions exist in the literature [16,6]. Two of those, known as flow and anti dependences, will be relevant for sliding.

Definition 3 (Flow Dependence). *A CFG node n is* flow dependent *on a CFG node m iff m defines a non-empty set of variables V that are used in n, i.e. $V \subseteq \mathrm{Def}(m) \cap \mathrm{Use}(n)$, and for any $v \in V$ there exists a path from m to n in the CFG with no further definition of v.*

Definition 4 (Anti Dependence). *A CFG node n is* anti dependent *on a CFG node m iff m uses a non-empty set of variables V that are defined in n (i.e. $V \subseteq \mathrm{Use}(m) \cap \mathrm{Def}(n)$, and for any $v \in V$ there exists a path from m to n in the CFG with no other definition of v.*

We will further stress that n is flow or anti dependent on m due to variables V. This will help us decide, later on, whether to consider a dependence when computing a co-slice.

Definition 5 (PDG). *The* program dependence graph *(PDG) corresponding to a given program's CFG is a labeled directed graph with the same nodes, N, n_{entry}, n_{exit}, as in the CFG, and with an edge (m, n, tag) directed from node m to node n iff n is control or data dependent on m. The value of $Kind(tag)$ is one of control,$flow$, or anti. The set $Vars(tag)$ denotes the variables contributing to a (flow or anti) data dependence.*

A subset of the PDG of the example program, including all flow- and control-dependence edges, is depicted on Fig. 5. PDG-based slicing, when started with a set of nodes C, finds all nodes from which there is a directed path of flow- or control-dependence edges to any $c \in C$ [6]. A PDG-based slice starting from the second definition of `index` at node 11, consists of the nodes $\{entry, 1, 2, 8, 9, 11\}$. One path causing the inclusion of node 1 goes through nodes 2 and 9, using a flow-dependence edge followed by two control-dependence edges.

Note that anti-dependence edges and the sets of variables causing data dependences are not used by slicing. Also, flow-dependence edges from the entry node are not significant, as the entry will be added to any non-empty slice due to control dependences. The anti-dependence edges, the labels on edges, and the flow-dependence edges from the entry node and to the exit are useful for the correct computation of a co-slice and for correctness checking. Figure 6 shows that portion of the PDG of the example code.

A traditional PDG does not include the exit node, as this node is not control dependent on any other node and does not use or define any variable. For sliding, however, it is helpful to assume $Use(exit)$ lists all variables that may be live on exit. This way, we get flow-dependence edges to the exit node from all final definition nodes of all results of the fragment represented by this PDG. A node n may define the final value of variable v if v is in $Def(n)$ and there exists a CFG path from n to the exit with no other definition of v. Since sliding extracts the slice of final values of a set of variables V, it will be convenient to start the slice from the exit node, after removing all edges to that node caused by other variables. (Similarly, a co-slice will be computed by slicing from the exit node after removing other edges.) For this to work, we must include all extracted variables V in $Use(exit)$, as well as all side effects on fields of objects that may be used outside. In Fig. 6, the local variables and object references in the example extracted sets V and V^+ are listed on the labels of flow-dependence edges to the exit node as optional, in square brackets, as they would not occur on the original PDG if it were not for the extraction.

A flow-dependence edge from the entry to the exit node lists all live-on-exit and extracted variables that may be modified but may also keep their initial value due to a CFG path from entry to exit with no definition of that variable. In the example, it is interesting to see that `index` is excluded from that edge, as its 5 definition nodes cover each potential path.

Definition 6 (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.*

Equivalently, note that in terms of a PDG with nodes N and edges E, including anti-dependence edges, a node n is a final-use node of v if and only if $v \in Use(n)$ and there exists no anti-dependence edge $(n, n', tag) \in E$ due to v, i.e. with $Kind(tag) = anti$ and $v \in Vars(tag)$.

3 PDG-Based Sliding

3.1 Source Code Considerations

Given a code fragment S and a set of variables V, a sliding transformation replaces S with a sequence of two sub-fragments, S_V and S_{CoV}, corresponding

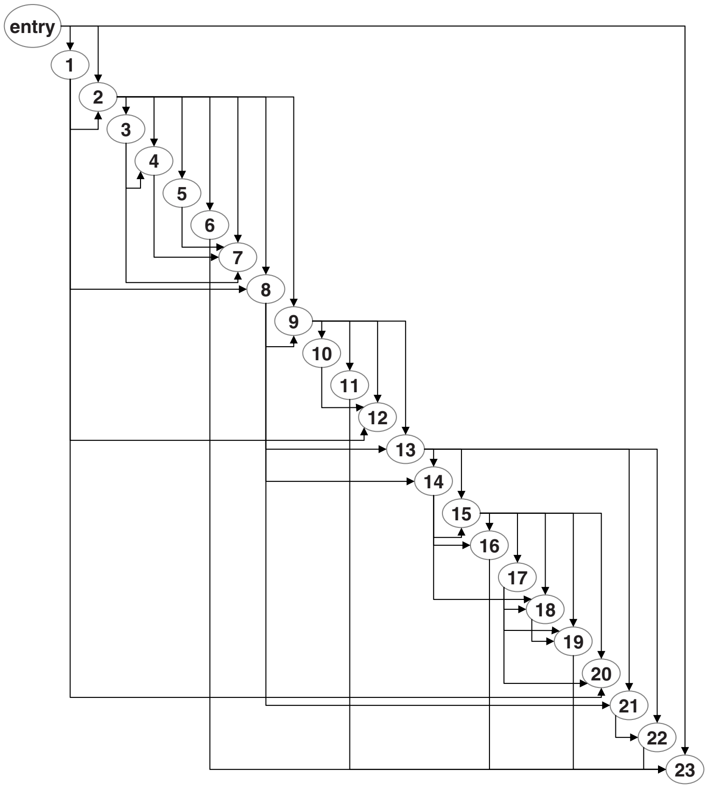


Fig. 5. A program dependence graph (PDG) representation of the example code from Fig. 1, with control- and data-dependence edges shown above and below the nodes, respectively

to the slice and co-slice of S on V, respectively. Suppose the fragment S is represented by a PDG, $(N, E, n_{entry}, n_{exit})$, as defined above. The algorithm in Fig. 7 accepts that PDG as input, along with the set of variables for extraction, V. It computes and returns a subset of nodes $N_V \subseteq N$ representing the slice, and another subset $N_{CoV} \subseteq N$ for the co-slice.

Sub-fragments S_V and S_{CoV} of S can be generated by removing all statements whose corresponding nodes are not present in N_V or N_{CoV}, respectively. We consider the removal of statements from blocks of code, even if nested in conditionals or loops. A more advanced sliding tool, supporting the decomposition of single statements too, would need to consider syntactic difficulties, and is left for future work. (For example, if the two side effects on parameters in `f(y++,z++);` are in the slice but the method call not, an extra semicolon would need to be added.) Two other syntactic considerations in the construction of S_V and S_{CoV} are related to local variable declarations and labeled blocks of statements. Those two constructs are not represented in the PDG by nodes. For the former, all declarations of vaiables not occuring in each side should be removed,

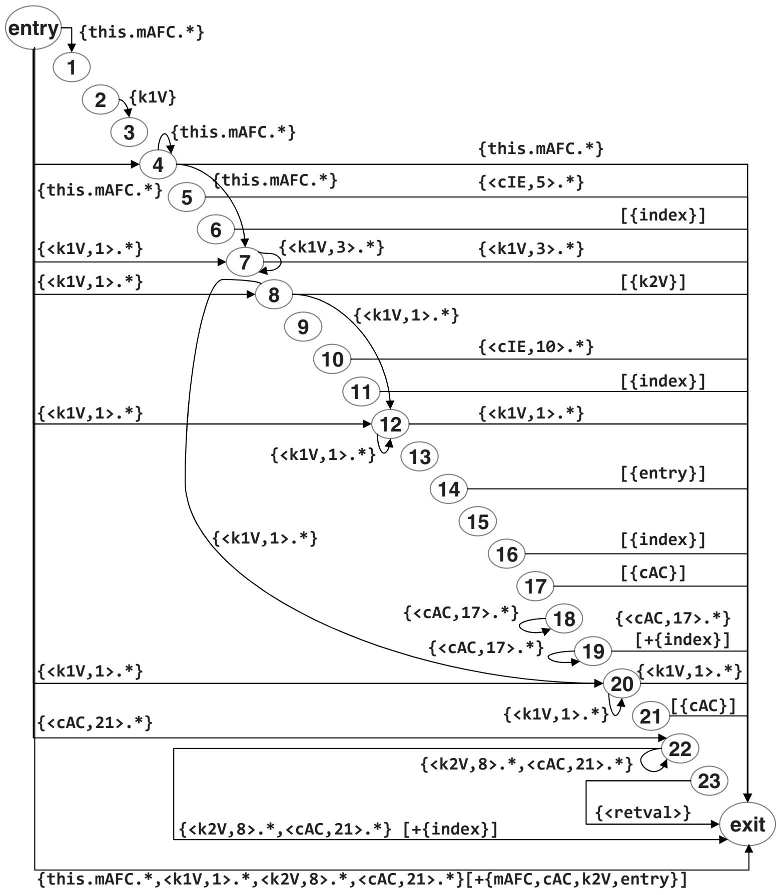


Fig. 6. A subset of the program dependence graph (PDG) representation of the example code from Fig. 1, showing flow-dependence edges from the entry node, to the exit node, as well as anti-dependence (curved) edges

and if a non-nested declaration occurs on both sides, its declaration should be removed from the co-slice, to avoid a compilation error. (An example of such declaration removal is shown on line 1 of Fig. 3). For the latter, if a non-nested block gets duplicated, its label should be renamed at least in one side.

When the functionality of the sequence of slice and its complement, in terms of the relation between input and output values for all live-on-exit variables, is guaranteed to be equivalent to that of the original code fragment, the successful sliding is considered a *compensation free* transformation. Otherwise, a variety of compensatory measures can be taken, e.g. in the form of variable localization and renaming. For this purpose, the PDG-based sliding algorithm of Fig. 7 computes and returns three further results, for the sets of potentially problematic variable instances $Pen1$, $Pen2$ and $Pen3$. Those instances may require compensation due

```
$ProgramSlidingOnThePDG(N, E, n_{entry}, n_{exit}, V)$
1  initialize the set of edges $FlowToExit_{NonV}$ to the empty set
2  **forall** $(m, n_{exit}, tag) \in E$ **do**
3      **if** $Vars(tag) \cap V = \emptyset$
4          add the edge $(m, n_{exit}, tag)$ to $FlowToExit_{NonV}$
5  $N_V := ComputeSlice(N, E \setminus FlowToExit_{NonV}, n_{exit})$
6  initialize $NonFinalUsesAtNode$ to map each $n \in N$ to the empty set (of variables)
7  **forall** $(m, n, tag) \in E$ **do**
8      **if** $Kind(tag)$ is $anti$
9          add all variables in $Vars(tag)$ to $NonFinalUsesAtNode(m)$
10 initialize the set of edges $FlowToFinalUse_V$ to the empty set
11 **forall** $(m, n, tag) \in E$ **do**
12     **if** $Kind(tag)$ is $flow$ and $Vars(tag) \subseteq (V \setminus NonFinalUsesAtNode(n))$
13         add $(m, n, tag)$ to the set of final-use edges $FlowToFinalUse_V$
14 $N_{CoV} := ComputeSlice(N, E \setminus FlowToFinalUse_V, n_{exit})$
15 $(Pen1, Pen2, Pen3) := CollectVariablesRequiringCompensation(N, E, V,$
16         $n_{entry}, N_V, N_{CoV}, NonFinalUsesAtNode)$
17 **return** $(N_V, N_{CoV}, Pen1, Pen2, Pen3)$
```

Fig. 7. A PDG-based sliding algorithm, collecting all nodes in the selected fragment's PDG, $(N, E, n_{entry}, n_{exit})$, belonging to the slice and the co-slice of the selected set of variables V. Three sets of variable instances potentially requiring compensation to avoid unintended data flow are computed too.

to definition of variables from V in the co-slice, non-final use of such variables in the co-slice, or definition in the slice of other (non-V) variables whose initial value may be needed in the co-slice. Some approaches to overcome the potential change in functionality are discussed in Sect. 3.4.

3.2 Computing the Slice

The proposed sliding algorithm works in three steps, for computing the slice, its complement, and finally checking correctness. The first step, on lines 1-5 of Fig. 7, computes the slice of final value of variables V, by starting from the exit node after having removed all flow-dependence edges (in the set $FlowToExit_{NonV}$) coming into the exit node due to variables not in V. In the example of $V = \{$`index`$\}$, the only remaining edges to the exit are the edges from the five definitions of `index`, on nodes {6,11,16,19,22}. Line 5 invokes the slicing algorithm of Fig. 8 on the exit node and the remaining edges, such that the first element n on line 4 of that algorithm will be the exit node, and all its remaining predecessors {6,11,16,19,22} will be added to *Slice* on the loop of lines 5-7, causing later addition of all nodes contributing to the final value of `index`. This way, the computation of S_V can be considered as having two substeps: first adding all predecessors of the exit node due to any member of V, then repeatedly adding all other nodes with a PDG path to those. In the example, the former substep operates on the PDG subgraph shown on Fig. 6, whereas the latter substep continues using the portion of the PDG shown on Fig. 5.

```
$ComputeSlice(N, E, c)$
1 initialize the result set of nodes $Slice$ to the singleton set $\{c\}$
2 similarly initialize $Worklist$ to $\{c\}$
3 while $Worklist$ is not empty do
4   take the first element $n$ out of the $Worklist$
5   forall $m \in N$ such that $(m, n, tag) \in E$ do
6     if $Kind(tag)$ is $flow$ or $control$ and $m \notin Slice$
7       add the PDG predecessor node $m$ of $n$ to $Slice$ and to $Worklist$
8 return $Slice$
```

Fig. 8. A traditional PDG-based slicing algorithm to collect all statement nodes from which there exists a directed path of dependence edges (flow or control, not anti) in the PDG, (N, E), ending in a node of interest c, known as the slicing criterion

To verify that the computed slices in the first two sliding examples are identical as expected, note that when sliding for the larger set V^+, compared with the sliding for V, further edges – such as $(8, exit)$ due to `entry` – survive the removal of $FlowToExit_{NonV}$. Still, since the source nodes of all those further edges are in the slice of `index`, these extra edges do not make the slice of V^+ larger or different. Indeed, in sliding for `index`, the edge $(8, exit)$ is removed from the PDG available for the slicing algorithm, but node 8 finds another way into that slice too. Since the edge $(11, exit)$ is not removed there, node 8 will be added to that slice due to, for instance, the dependence path $< 8, 9, 11 >$ (see Fig.5).

3.3 Computing the Co-slice

The next step, on lines 6-14 of Fig.7, is co-slicing. It involves slicing from the exit node again (line 14), but for variables outside the set V this time. To maximize reuse, the removal of redundant flow-dependence edges (lines 10-13) is not restricted to edges leading to the exit node. A flow-dependence edge $(l, m, tag) \in E$ can be removed if the variables causing it, $Vars(tag)$, form a subset of V, and the node it ends in, m is a final-use node for all those variables. Node m is guaranteed to be a final-use node of variable v if v does not contribute to any anti-dependence edge from m, i.e. for any $(m, n, tag') \in E$, we get $v \notin Vars(tag')$.

Since the sets of final-use variables at each node will be needed later in checking for correctness (lines 15-16 and the called algorithm of Fig. 9), a separate stage (lines 6-9) computes and stores this information for all nodes. For convenience (and efficiency), it maps each node to the set of *non*-final uses; it first assumes no variable is a non-final use at all nodes (line 6) and then whenever evidence is found for a non-final use, through an anti-dependence edge from a node m, it adds to the set of non-final uses of m (on line 9) all the variables this edge is due to. Notably, the use of variables on the exit node is certainly a use of their final value; this is confirmed by the lack of anti-dependence edges from the exit.

In the example, when preparing to compute the co-slice of V, all edges from the definition nodes of `index`, being from $\{6, 11, 16, 19, 22\}$ to the return statement's node, 23, are added to $FlowToFinalUse_V$ (on line 13) and hence removed from the co-slice calculation. This is so because none of those definition nodes can be reached (in terms of control flow) from the return node. Accordingly, there exists no anti-dependence edge leaving node 23 (see Fig. 6), so we never get to line 9 of the algorithm with m being 23, hence the set $NonFinalUsesAtNode(23)$ remains empty (as initialized on line 6). More tricky are the edges from those final-definitions of `index`, nodes $\{6, 11, 16, 19, 22\}$, to the exit node. The edges from the last two, $(19, exit)$ and $(22, exit)$ are due to the sets `{<cAC,17>.*,index}` and `{<k2V,8>.*,<cAC,21>.*,index}`, respectively (Fig. 6). Since those are not subsets of the set V (i.e. the singleton `{index}`), line 13 of the algorithm is not reached, and those two edges are not removed. Indeed the co-slice of V, on Fig. 3, includes statements 19 and 22 due to their side effects on the state of the objects on which the method `putIfAbsent()` is invoked. When sliding for V^+, the object fields `{<cAC,17>.*,<k2V,8>.*,<cAC,21>.*}` are included in the set of extracted variables, and hence the edges $(19, exit)$ and $(22, exit)$ are added to $FlowToFinalUse_V$ (on line 13) such that nodes 19 and 22 are not added to the co-slice (see bottom part of Fig. 2).

Note that had node 19 been added to the co-slice of V^+, node 18 would have been added too. The flow-dependence edge from node 18 to 19 (due to `<cAC,17>.*` which is in V^+) is not added to $FlowToFinalUse_V$ (on line 13), because of the anti-dependence edge $(19, 19)$, which is due to `<cAC,17>.*` (Fig. 6). In contrast to the redundancy of self dependence edges in the context of slicing, this example shows how self anti-dependence edges are relevant for co-slicing. They reflect the fact that a variable is both used and defined in a single statement, such that its value on exit from the statement may differ from its value on entry.

3.4 Correctness Checking and Compensation

The final step of the proposed PDG-based sliding algorithm involves checking for correctness of the transformation, in terms of preserving functionality. In Fig. 7, the checking procedure is invoked on lines 15-16. This procedure, detailed on Fig. 9, involves the collection of three sets of potentially problematic variable instances. When all three sets are empty, the replacement of the original code fragment with the computed slice followed by the co-slice, is guaranteed to preserve the original functionality, such that for a given input state on which the original fragment terminates, the transformed fragment would terminate too, leaving the program in the same state as the original in all variables.

Let's examine the variables for extraction V first. The responsibility for computing the correct value for those variables lies on the slicing algorithm and the input PDG it is computed on. Assuming the slice computes the expected value for each $v \in V$, we need to make sure this correct value is maintained by the co-slice. The simplest way to ensure this is to check that v is not in the set of defined variables of any node of the co-slice, $n \in N_{CoV}$. The set $Pen1$ collects those problematic variable instances (lines 3-4). In the example of Fig. 2 this set is empty. In the example of Fig. 3, in turn, we get two instances of `index` in this set, for its definitions in nodes 19 and 22. These nodes are included in the co-slice not due to modifying `index`, but rather due to relevant side effects on live-on-exit object fields.

In terms of compensation in the co-slice, our choice on Fig. 3 was to remove the impact on the offending member of V, `index`, by removing the assignments to it. This was simple to do in this case but may be more challenging in others. For example, if the undesired definition takes place in a called method, changing its code may require the duplication of that method, in case its original version is still called from elsewhere. The flavor of sliding proposed in this paper is restricted to a local transformation of the selected fragment. Instead, if the value of that variable is accessible at the beginning of the transformed code fragment, it can be backed up in a local variable, ahead of the slice, and restored ahead of the co-slice. (Accessibility may be problematic, for example, when the variable is a private field of a different class.)

When the unwanted definition is performed in the selected fragment and is tricky to eliminate, say due to syntactic difficulties (as mentioned earlier), a preferred alternative would be to localize the effect by adding a new variable and updating the problematic instances to refer to the new variable. To enable this corrective measure, pairs of node and variable name are collected and returned by the checking procedure, instead of just variable names.

Moving on to examine all the other variables relevant (i.e. live) on exit from the original fragment, note that, by construction, the live-on-exit variables are $\mathit{Use}(n_{exit})$. We therefore need to consider variables in the set $\mathit{CoV} := \mathit{Use}(n_{exit}) \setminus V$.

The co-slice is computed as a slice of final values of variables CoV, using final values of V where it is guaranteed to be correct to do so. On entry to the co-slice, we can assume to have the expected final value of each variable $v \in V$ available. For correct operation of that co-slice, we first need to ensure no other (i.e non-final) value of a variable in V is used in any node $n \in N_{CoV}$. Such instances are collected in the set $\mathit{Pen}2$ (lines 5-6). In both earlier examples, all uses of variables V and V^+ are final. Sliding for the local object reference `key1Value`, whose slice consists of nodes 1,2, and 3, would yield a co-slice with a non-final use of that variable on node 2 – evidenced by the anti-dependence edge $(2, 3)$ on Fig. 6.

In terms of compensation, the non-final use of a variable v requires the addition of a new variable, v', and renaming of those instances of the non-final use, in the co-slice. When the initial value of v is needed in the co-slice, it should be backed up ahead of the slice, if possible. Again, the preparation of such a backup might not be possible if the variable is not local and not accessible at the head of the transformed fragment. Also, renaming might not be possible if v is not local and used in a called method. (Note that the set of variable instances $\mathit{Pen}2$ will specify a node n where the non-final use took place, but this may be a node involving a method call, and v may be an object field referred to inside this method, when the object reference is passed explicitly as a parameter, not the field v itself.)

In the example of sliding for `key1Value`, were node 2 of the co-slice would include a non-final use of that variable, localization would work well, with no neeed for backup of the initial value, due to the initialization of `key1Value` on the statement of node 1 (see Fig. 1).

Finally, for all other initial values demanded by the co-slice, outside of the set V, we must ensure they hold their initial value on entry to the co-slice. We first collect all non-V variables that may require an initial value at the co-slice, in the set $InputToCoSliceNonV$ (lines 7-11). Those are the non-V variables contributing to a flow-dependence edge from the entry to any node $n \in N_{CoV}$. Then, the simplest way to ensure those variables hold the initial value on entry to the co-slice is by checking that they are not defined in the slice nodes N_V. Definitions of all members of $InputToCoSliceNonV$ in any slice node are recorded in the set $Pen3$ (lines 12-14). In the example of Fig. 3, for sliding `index`, we get the input to the duplicated node 22 in the co-slice, `<cAC,21>.*`, also defined in the slice on that same node (see labels on flow-dependence edges $(entry, 22)$ and $(22, exit)$ on Fig. 6).

A preferred form of compensation, again, would be to localize the unwanted effects in the slice, if possible. When the initial value is not needed there, and the definition is explicit (i.e. not performed inside a called method), this would be a simple addition of a local variable in the slice.

In the mentioned example, the effects on fields of the object referred to by the local variable `cAC` (i.e. `charArrayCache` in Fig. 3), are due to implicit definitions on node 22. The actual definitions take place in the method `putIfAbsent()` of the class `CharArrayCache`, so simple renaming better be avoided as it would require duplication of that method. Now, in this case the initial value of those fields is needed for correct operation of the slice, as can be witnessed by the edge $(entry, 22)$ on Fig. 6. And as it turns out, it is possible to back up those fields, since they are not private and the classes are in the same package. Still, it is not clear that this would be an acceptable practice. Alternatively, one could consider making a backup of the entire object referred to by `charArrayCache`. This would require the object to be cloneable, and it is arguable that this kind of compensation would be acceptable too. Note that such a backup would not be trivial to prepare ahead of the slice, as the local object reference `charArrayCache` is computed in the slice itself.

The measures of compensation taken in the experiment of Sect. 5 below involve variable localization and local backups, but no cloning of objects.

4 Sliding-Based Refactoring

When applied to source code, sliding can be considered a refactoring by itself, as it may improve the structure of existing code, making it more readable and easier to maintain. Naturally, a sliding transformation can be followed up by steps of method extraction, as the slice and its complement are made contiguous and hence ready for extraction by the Extract Method refactoring [7].

```
$CollectVariablesRequiringCompensation(N, E, V,$
        $n_{entry}, N_V, N_{CoV}, NonFinalUsesAtNode)$
1   initialize $Pen1$,$Pen2$, and $Pen3$ to empty sets
2   **forall** $n \in N_{CoV}$ **do**
3       **forall** $v \in Def(n) \cap V$ **do**
4           add the pair $(n, v)$ to $Pen1$
5       **forall** $v \in Use(n) \cap V \cap NonFinalUsesAtNode(n)$ **do**
6           add the pair $(n, v)$ to $Pen2$
7   initialize $InputToCoSliceNonV$ to the empty set
8   **forall** $n \in N_{CoV}$ **do**
9       **forall** $(n_{entry}, n, tag) \in E$ **do**
10          **if** $Kind(tag)$ is $flow$
11              add all variables in $Vars(tag) \setminus V$ to $InputToCoSliceNonV$
12  **forall** $n \in N_V$ **do**
13      **forall** $v \in Def(n) \cap InputToCoSliceNonV$ **do**
14          add the pair $(n, v)$ to $Pen3$
15  **return** $(Pen1, Pen2, Pen3)$
```

Fig. 9. Analysis of the slice and co-slice to identify potential unintended flow of data if the sequence of slice and co-slice were to replace the original code

This section revisits three refactoring techniques from existing catalogs [7,1]: Split Loop, Replace Temp with Query, and Separate Query from Modifier. A revision of the mechanical description of how to perform the refactoring transformation is proposed, with sliding being a key step in all three cases. The revised mechanics, being more concrete and constructive than the original descriptions, contribute to the applicability of the refactorings, making them more amenable for future automation.

A direct application of sliding is the Split Loop refactoring [1]. Fig. 10 shows its proposed mechanics. The user can simply follow the sliding algorithm, or use a tool, and select the variables of interest for extraction. If the initialization of the loop index is not in the selected code fragment, a tool based on the compensation-free flavor of sliding, expecting empty sets in the resulting $Pen1$,$Pen2$, and $Pen3$ sets (see Fig. 9), 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. Another advanced sliding implementation would avoid some code duplication by identifying all final values of the loop whose computation is fully included in the extracted slice. The addition of those variables to the set of variables for extraction, V, would cause the addition of further edges to the set $FlowToFinalUse_V$ (see lines 10-13 in the sliding algorithm of Fig. 7). Those added flow-dependence edges, directed from final-value definition nodes to the exit, are then removed from consideration when computing the co-slice, potentially making the co-slice (i.e. second instance of the loop) smaller. The value of a smaller co-slice is not only in the reduced levels

Mechanics for Split Loop

- Perform sliding on the loop fragment choosing a subset V of the loop's results.
    - *Avoid unnecessary code duplication by adding to V all loop results whose addition will not increase the size of the first resulting loop.*
    - *If sliding fails update the code to avoid the failure and repeat this step, or choose a different loop to split.*
- Compile and test.

Fig. 10. Sliding-based mechanics for Split Loop

of code duplication, but also in the potential need for less compensation, as can be witnessed by the collection of $Pen1$ and $Pen2$ on lines 2-6 of Fig. 9.

New mechanics for the Replace Temp with Query (RTwQ) refactoring [7] are presented on Fig. 11. This refactoring involves the extraction of the computation of a single variable, a temporary one, into a method of its own. That method should 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. Applying RTwQ twice, on the variables `def` and `pot` of Fig. 12, would replace their final use in line 2009 with calls to the new methods, on line 1995 of the resulting code, on Fig. 13. There, the two new methods, `def()` and `pot()`, can be seen starting on lines 2001 and 2016, respectively.

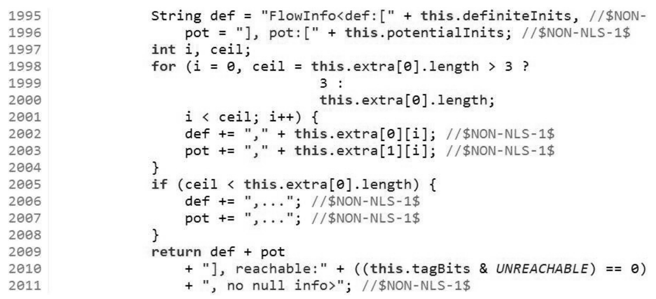
Note that the loop has been eliminated from the original code, in this example. Interestingly, one more loop in the same original method included the same computation of `def` and `pot`. In that other loop (not shown here), the two computations were entangled with a computation of one other value (`nullS`, whose final use can be seen on line 1983 on Fig. 13. Subsequent applications of RTwQ on that other loop, for `def` and `pot` again, would replace their final use with calls to the previously extracted methods. Those two invocations can be seen on line 1981 of the resulting code on Fig. 13. This is an example where the initial RTwQ caused duplication of the loop, yet it has enabled further refactoring steps to reduce duplication through the elimination of non-trivial code clones.

A third refactoring that can benefit from sliding is Separate Query from Modifier (SQfM) [7]. 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 value, the one returned from the original method. Note that this value will not necessarily reside in a single variable since the result of some expression could be returned, and since multiple return statements may refer to different variables. Accordingly, an optional first step, in preparation for sliding, is dedicated to the introduction of a local variable to hold the returned result. When the code includes more than one return statement, each return is replaced by an assignment to the added variable and a jump to the end, where a single return of that variable is inserted. In Java, the jump can be implemented through a break from

Mechanics for Replace Temp with Query

- Identify the relevant fragment of code, from the temp's declaration to the end of its enclosing block.
- Perform sliding on that code fragment for the selected temp.
  - *If the sliding fails you may want to choose a different temp to replace with a query.*
  - *Successful sliding will bring together the code for computing the final value of the temp, making it a contiguous fragment ready to be extracted into a method of its own.*
- Compile and test.
- Perform Extract Method on the extracted slice, giving it an appropriate name.
  - *If the extracted method appears to have side effects consider extracting a different temp, or modify the code to prevent the side effects.*
  - *If all those side effects occur in invoked methods, consider applying Separate Query from Modifier on those methods before re-applying this refactoring. That way, the extracted modifiers could possibly be excluded from the query, and cause no side effects.*
- Compile and test.
- Perform Inline Temp on the selected temp at its declaration.
  - *This step involves the replacement of all references to the temp with the call to the extracted method (i.e. the query), and the removal of the temp's declaration.*
  - *If the value of any of the query's parameters may be different at any point of reference, consider adding backup variables at the temp's point of declaration, or abandon the refactoring.*
  - *A potential cause of failure is the need of backup for a non-cloneable parameter object; making such backup might otherwise not be desirable due to space (i.e. large object to clone) or other considerations.*
- Compile and test.

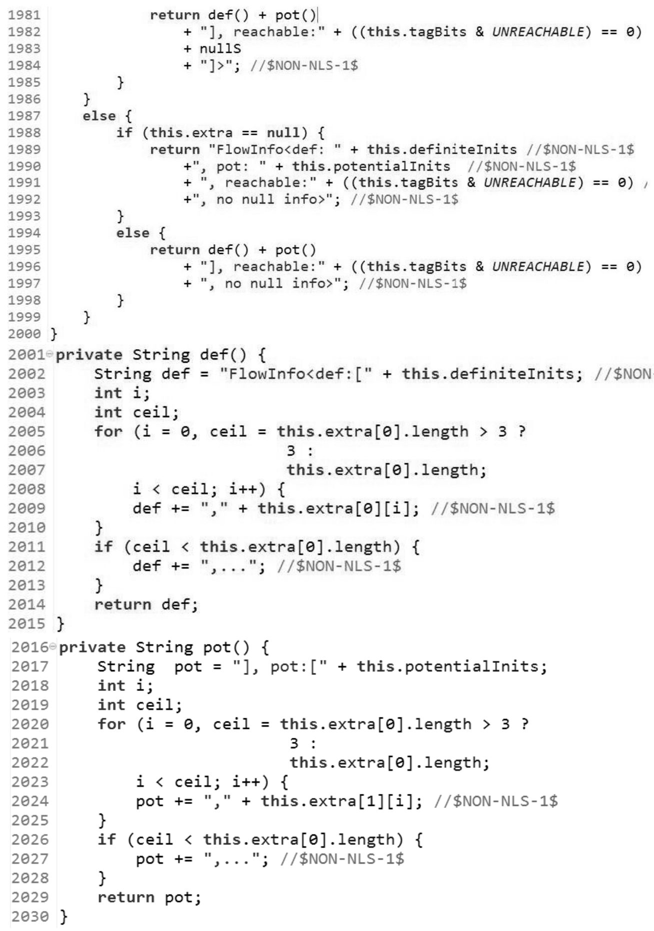
Fig. 11. Sliding-based mechanics for Replace Temp with Query



```
String def = "FlowInfo<def:[" + this.definiteInits, //$NON-
    pot = "], pot:[" + this.potentialInits; //$NON-NLS-1$
int i, ceil;
for (i = 0, ceil = this.extra[0].length > 3 ?
                3 :
                this.extra[0].length;
    i < ceil; i++) {
    def += "," + this.extra[0][i]; //$NON-NLS-1$
    pot += "," + this.extra[1][i]; //$NON-NLS-1$
}
if (ceil < this.extra[0].length) {
    def += ",..."; //$NON-NLS-1$
    pot += ",..."; //$NON-NLS-1$
}
return def + pot
    + "], reachable:" + ((this.tagBits & UNREACHABLE) == 0)
    + ", no null info>"; //$NON-NLS-1$
```

Fig. 12. Example RTwQ, original code

a labeled block. This preparatory step should be undone before the end of the refactoring, when the labeled block and result variable are located in designated methods for the query and modifier, and can be replaced with return statements.



```
            return def() + pot()
                + "], reachable:" + ((this.tagBits & UNREACHABLE) == 0)
                + nullS
                + "]>"; //$NON-NLS-1$
        }
    }
    else {
        if (this.extra == null) {
            return "FlowInfo<def: " + this.definiteInits //$NON-NLS-1$
                +", pot: " + this.potentialInits  //$NON-NLS-1$
                + ", reachable:" + ((this.tagBits & UNREACHABLE) == 0) /
                +", no null info>"; //$NON-NLS-1$
        }
        else {
            return def() + pot()
                + "], reachable:" + ((this.tagBits & UNREACHABLE) == 0)
                + ", no null info>"; //$NON-NLS-1$
        }
    }
}
private String def() {
    String def = "FlowInfo<def:[" + this.definiteInits; //$NON-
    int i;
    int ceil;
    for (i = 0, ceil = this.extra[0].length > 3 ?
                        3 :
                        this.extra[0].length;
        i < ceil; i++) {
        def += "," + this.extra[0][i]; //$NON-NLS-1$
    }
    if (ceil < this.extra[0].length) {
        def += ",..."; //$NON-NLS-1$
    }
    return def;
}
private String pot() {
    String  pot = "], pot:[" + this.potentialInits;
    int i;
    int ceil;
    for (i = 0, ceil = this.extra[0].length > 3 ?
                        3 :
                        this.extra[0].length;
        i < ceil; i++) {
        pot += "," + this.extra[1][i]; //$NON-NLS-1$
    }
    if (ceil < this.extra[0].length) {
        pot += ",..."; //$NON-NLS-1$
    }
    return pot;
}
```

Fig. 13. Example RTwQ, after replacement of two temporary variables with queries

After sliding the computation of this returned value away from the remaining computations (i.e. the code with the side effects), we perform Extract Method twice, on the slice and co-slice, followed by the optional undoing of the preparatory step. In the final step, we perform Inline Method, to replace all calls to the original method with the two invocations, of the query and modifier.

Beyond its immediate application as a refactoring, where the transformed code is a candidate for further development and therefore might be committed at a subsequent code change delivery, the SQfM transformation can be useful also for temporarily updating the code in a testing, debugging, or verification scenario. Such usage of SQfM might enable the application of tools and techniques that assume no side effects exist in conditional expressions.

Proposed mechanics for SQfM are presented in Fig. 14. The sliding of `index` on the code from Fig. 1, yielding the slice shown on the top of Fig. 2 and the

Mechanics for Separate Query from Modifier

- Prepare the method for sliding by ensuring it has a single return statement.
    - *Enclose all statements in the method body in a labeled block.*
    - *Insert a declaration above the added block for a new temporary variable to store the method's result, and a statement to return its final value below that block.*
    - *Modify all return statements in the labeled block to store the returned value and to break to the added label (i.e. out of the method's body through the added return statement).*
    - *This preparatory step may be skipped when the original method is already constructed with a single retrun of a single variable.*
- Compile and test.
- Perform sliding on the labeled block for the temp designated in the first step above.
    - *If sliding fails choose a different method to separate.*
- Compile and test.
- To construct the query, perform Extract Method on the slice, giving it an appropriate name.
    - *If the slice appears to have side effects consider the separation of a different method, or modify the code to prevent the side effects.*
    - *If all those side effects occur in invoked methods consider applying this refactoring on those methods before re-applying it on the present method. That way, the extracted modifiers could possibly be excluded from the query, and cause no side effects.*
- Compile and test.
- Perform Inline Temp on the designated temp if you prefer to have the query called after the modifier, possibly also inside the modifier if its result is used there.
- To construct the modifier, perform Extract Method on the updated co-slice, giving it an appropriate name.
- In the query and modifier methods, undo the preparatory step by re-introducing the original return statements, removing the added temp, break statements, and labeled block.
    - *Take care when re-introducing return statements in the (now void) modifier method, to avoid re-introducing the returned value.*
- Compile and test.
- Perform Inline Method on the refactored version of the selected method
    - *This will replace all calls to the original method with calls to the query and modifier. Note, however, that at each call site the query or modifier might be redundant, if the respective results are never used.*
    - *If the method is part of some inheritance relation (i.e. overriding a method in a superclass, being overriden in a subclass, or implementing a method declared in an interface), consider performing this refactoring throughout the hierarchy. Alternatively, consider skipping this step in such cases, leaving the calls to the extracted query and modifier in the original method.*
- Compile and test.

Fig. 14. Sliding-based mechanics for Separate Query from Modifier

co-slice shown on Fig. 3, is an example step toward SQfM. Successful completion of SQfM, in this case, would require a preliminary SQfM step to split the method `putIfAbsent()`, called on lines 19 and 22. This way, the slice would include a call to the query, contributing to the computation of `index`, whereas the co-slice would call the modifier, for effecting the fields of `charArrayCache` as needed.

5 Evaluation

To evaluate the potential of sliding for supporting refactorings, as presented above, in terms of the number of successful cases, the ability to compensate when the need arises, and the levels of code duplication, a preliminary experimentation to examine 55 cases has been performed, manually, on real Java code. The subject project has been the Java compiler in Eclipse. The comprehensive test suite of this compiler, featuring nearly 70,000 automated tests, with over 40,000 regression test cases, may provide us with some confidence regarding the correctness of the transformation steps.

As candidate sliding criteria, two kinds of slices were considered for extraction: slices with at least one partial loop, and slices of the value returned from a non-pure function. Isolation of the former exercises loop untangling through the Replace Temp with Query refactoring, while the extraction of the latter provides separation of commands from queries, exercising the Separate Query from Modifier refactoring.

The candidate criteria were identified as follows. For RTwQ, the Extract Method refactoring tool in Eclipse was employed on each loop, looking for rejected cases due to “ambiguous result”. Such an error message is issued by the tool whenever more than one local variable is updated in the selected fragment (i.e. in the loop) and is live on exit from that fragment. A selection of 8 loops was examined. Those loops were found in 7 different methods of 6 different classes, with 2 local results in 4 cases and 3 results in the other 4 cases. The sliding and subsequent steps of RTwQ were performed on each local result, in sequence, with the fragment S being the full scope of the variable's declaration. When performing the extraction in a different order presented an important difference in the results, the alternative order was investigated too. In total, 23 cases of sliding for RTwQ were recorded this way.

For SQfM, any non-void method with side effects is a potential candidate, and 32 such cases, found in a package named `org.eclipse.jdt.internal.compiler.codegen` were examined.[1] All examples in this paper were taken from this experiment.

The resulting code, after the sliding step and the complete refactoring, in the successful cases, was tested and passed all tests. For sanity checking, deliberate mistakes were added to see that the code is indeed tested. Only one of the 38 succssfully refactored methods was not exercised by any test.

[1] Thanks to Alex Libov for automating the identification of both RTwQ and SQfM candidates, using Eclipse's JDT, its refactoring API, and the side effect analysis called *ModRef* in WALA.

In a preliminary step, the source code of many of the subject methods needed to be updated, to remove side effects from expressions, such as assignments within the predicates of `if` statements. The manual step would isolate the assignment into its own statement. This was not done exhaustively, but rather on demand, when the slice involved only the result of the expression or only the side effect. A future sliding algorithm and tool should better treat such cases correctly, without the need for manual update. Another type of manual change was to replace an early return statement with a break statement from a labeled block, as explained in Sect. 4 above. The change was later undone, after extraction of the slice and co-slice into new methods. One other type of manual change was to employ the SQfM refactoring on a called method in cases where the slice required only the result or only the side effect. And one final type of change was to move the declaration of a local variable, or to break two declarations in one statement into two separate declarations. In total, for 54 cases of sliding, a total of 77 manual changes were performed. (The total is 54, not 55 cases, due to the success in only 31 of the 32 SQfM cases, as will be explained below, added to the 23 RTwQ cases.)

The size of the code fragments ranged from 6 to 97 (in a method `numberOfDifferentLocals()` of class `StackMapFrame`), with an average of 23.5 statements per case. The slice size ranged from 1 statement (a return statement, returning a constant) to 85 (in the 97-statement method), averaging 10.4 statements. The co-slice size ranged from 1 to 82 (again in the same largest method) with an average of 20 statements, leading to an average of 6.9 duplicated statements per sliding.

In terms of the need for compensation, over 44% (7 RTwQ cases and 17 cases of SQfM) required no compensation at all. For all the remaining cases, variable localization and the introduction of backup variables (as discussed in Sect. 3.4 above) of at most 3 locals or fields per sliding case proved sufficient.

In terms of reuse, 132 out of the 239 total uses of an extracted variable were uses of a final value, averaging some 2.4 final uses and 1.99 non-final uses per case. After sliding, 46 non-final uses were left in the co-slices, leaving us with some 0.85 non-final uses per case.

The single most problematic case for SQfM, in class `ConstantPool`, involves the following code:

```
public byte[] dumpBytes() {
    System.arraycopy(this.poolContent, 0, (this.poolContent =
        new byte[this.currentOffset]), 0, this.currentOffset);
    return this.poolContent;
}
```

Sliding of the result without reuse of the allocated field array would fail, in this case, returning a reference to a different object than the field (with equal value). Another problem is that localization of the field array would require copying its initial value. A different type of separation is required here, possibly to compute the new field first, and only then return the result stored in that field.

6 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 [3,13]. 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 [15,8] considered the extraction of contiguous code only.

Maruyama [14] 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 reuse of the extracted result was not of the final value only, but of any value defined by that variable. This way, the co-slice may make a reference intended for a non-final value, but get to use, instead, the final extracted value, making the transformation incorrect. This incorrectness was reported by Tsantalis et al. [17]; their more recent work constructs the complementary code in the same way, but defines PDG-based rules to identify these problematic cases and reject the transformation.

A number of provably correct algorithms for the extraction of a set of not-necessarily contiguous statements have been proposed in the literature [12,11,10].

Of those, tucking [12] 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 a fragment that encloses 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 [10], 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 loops as in our example of `def` and `pot`, 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 RTwQ example of `def` and `pot` [5]. 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 of extracting the slice of `index` (see Fig. 1) from the full fragment of 1-23, that complement would include the whole fragment, as statement 23 would be included in the co-slice and then cause all the slice to be duplicated.

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 [11,10].

7 Conclusion

To paraphrase Weiser's seminal work [19], 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 wish to thank Aharon Abadi, Yishai Feldman, Michiaki Tatsubori, and Shmuel Tyszberowicz for intriguing discussion and comments, and the anonymous reviewers for their helpful suggestions, all directing me at improvements to the paper.

My special thanks go to Cindy Eisner, Steve Fink, and Maayan Goldstein for the encouragement and crucial advice at key decision points.

References

1. An online refactoring catalog, `http://www.refactoring.com/catalog/`
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1988)
3. Cimitile, A., Lucia, A.D., Munro, M.: Identifying reusable functions using specification driven program slicing: a case study. In: ICSM, pp. 124–133 (1995)
4. Ettinger, R.: Refactoring via Program Slicing and Sliding. Ph.D. thesis, University of Oxford, Oxford, United Kingdom (2006)
5. Ettinger, R., Verbaere, M.: Untangling: a slice extraction refactoring. In: AOSD 2004: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, pp. 93–101. ACM Press, New York (2004)
6. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987), `http://doi.acm.org/10.1145/24039.24041`
7. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison Wesley (2000)
8. Griswold, W., Notkin, D.: Automated assistance for program restructuring. ACM Transactions on Software Engineering 2(3), 228–269 (1993)
9. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. 12(1), 26–60 (1990)
10. Komondoor, R., Horwitz, S.: Effective automatic procedure extraction. In: Proceedings of the 11th IEEE International Workshop on Program Comprehension (2003)
11. Komondoor, R., Horwitz, S.: Semantics-preserving procedure extraction. In: POPL 2000: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 155–169. ACM Press, New York (2000)

12. Lakhotia, A., Deprez, J.C.: Restructuring programs by tucking statements into functions. Information and Software Technology 40(11-12), 677–690 (1998), `citeseer.nj.nec.com/lakhotia99restructuring.html`
13. Lanubile, F., Visaggio, G.: Extracting reusable functions by flow graph-based program slicing. IEEE Trans. Software Eng. 23(4), 246–259 (1997)
14. Maruyama, K.: Automated method-extraction refactoring by using block-based slicing, pp. 31–40. ACM Press (2001)
15. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, IL, USA (1992), `citeseer.nj.nec.com/opdyke92refactoring.html`
16. Ottenstein, K., Ottenstein, L.: The program dependence graph in a software development environment. In: Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp. 177–184 (1984)
17. Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities. In: CSMR 2009: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, pp. 119–128. IEEE Computer Society, Washington, DC (2009)
18. Verbaere, M., Ettinger, R., de Moor, O.: JunGL: a scripting language for refactoring. In: ICSE, pp. 172–181 (2006)
19. Weiser, M.: Program slicing. In: ICSE, pp. 439–449 (1981)