ACADEMIC COLLEGE OF TEL AVIV-YAFFO

MASTER THESIS

An efficient syntax-preserving slide-based algorithm for program slicing

Author:
Adi MARINOV

Supervisors:
Prof. Shmuel TYSZBEROWICZ
and Dr. Ran ETTINGER

A thesis submitted in fulfillment of the requirements for the degree of Master of Computer Science

March 3, 2020

ACADEMIC COLLEGE OF TEL AVIV-YAFFO

Abstract

Computer Science
School of Computer Science

Master of Computer Science

An efficient syntax-preserving slide-based algorithm for program slicing

by Adi MARINOV

Software systems continuously evolve over the years to avoid becoming less useful. However, evolution causes the code to diverge from the original design. Thus, the internal structure may change and reduce design quality. Refactoring is one of the major approaches that assist evolving software, yet keeping high design quality. The Extract Method refactoring enables to move a code fragment that can be grouped together into a separated method, replacing the old code with a call to the new method. For that, Program Slicing can be used.

In this thesis we present a backward, static, syntax-preserving slicing algorithm that elaborates on the Static single assignment form (SSA)-based slicing algorithm described in the PhD thesis of Ettinger. The slicing algorithm for program P involves 3 steps: converting P into an SSA form; computing its flow-insensitive slice; converting it back from SSA. The algorithm requires time polynomial in the size of the program. We defined a new slicing algorithm that is asymptotically faster and easier to implement than the SSA-based algorithm.

Acknowledgements

I would like to express my sincere gratitude to my thesis advisers, Prof. Shmuel Tyszberowicz and Dr. Ran Ettinger.

The door to Prof. Tyszberowicz's office was always open for any question or problem I faced, even for a quick chat full of insightful comments. Dr. Ettinger taught me never to give up until I reach perfection and our meetings always made me extremely motivated moving forward with my research. I thank you both for your guidance and inspiration.

Finally, I would like to thank my husband Tomer and my son Yuval for their patience, full support and encouragement throughout the process of my research.

Contents

Abstract **i**

Acknowledgements **ii**

1 Introduction **1**
1.1 Goal of this thesis . . . 1
1.2 Background . . . 2
1.2.1 Program slicing . . . 2
1.2.2 Programming notations and representation . . . 5
1.2.3 Data flow analysis . . . 5
1.2.4 Slips and slides . . . 7
1.2.5 Slide dependence . . . 8
1.2.6 Dafny . . . 9
1.2.7 Static single assignment . . . 9
1.2.8 SSA-based slicing . . . 10
1.3 Contributions . . . 13

2 Formal framework **14**
2.1 Running example . . . 14
2.2 Programming notations and representation . . . 16
2.3 Program analysis . . . 18
2.3.1 Reaching definitions . . . 18
2.3.2 Liveness . . . 20
2.4 Slides . . . 21
2.4.1 Slide dependence graph . . . 22
2.4.2 Paths in a slide dependence graph . . . 23
2.5 VarSlides . . . 24
2.5.1 VarSlide dependence graph . . . 24
2.5.2 Paths in a varSlide dependence graph . . . 25
2.6 Correspondence between slideDG and varSlideDG . . . 26

3 Properties of slide dependence graphs **28**
3.1 Reaching definitions and liveness . . . 28
3.2 Reachability in the slideDG and the varSlideDG . . . 30

4 A novel slide-based slicing algorithm **35**
4.1 The algorithm . . . 35
4.2 Examples . . . 35

5 Proof of correctness **38**
5.1 Our algorithm . . . 38

6 Conclusion **43**

A Appendix **45**
A.1 Full definitions 45
A.1.1 Utility functions 45
A.1.2 Slides functions 46
A.1.3 VarSlides functions 47
A.1.4 Graphs correspondence functions 49
A.2 Full proofs of theorems and lemmas 52
A.2.1 Reaching Definitions and Liveness for exit 53
A.2.2 Reaching Definitions and Liveness for entry 56
A.2.3 Identical Skip Slips 59
A.2.4 Inverse varLabelOf 61

Bibliography **65**

Chapter 1

Introduction

Software maintenance is the most expensive activity in the software lifecycle. This process includes the following activities: adding new features and deletion of obsolete code (perfective), correcting errors (corrective), adapting to new environments (adaptive), and improving code quality (preventive) [11]. The latter increases software maintainability in order to prevent future problems. Software systems continuously evolve over the years to avoid becoming less useful [15]. However, software evolution causes the code to diverge from the original design, since code is often modified without referring to the effect of those modifications on the design. Hence, code eventually loses its structure and it becomes difficult to add new features or to modify the code without introducing new bugs [15]. Thus, even when starting with high quality of design (e.g., low coupling and high cohesion) the internal structure may change and reduce the design quality. Refactoring is one of the major approaches that assist evolving software, yet keeping high design quality [10].

Refactoring is defined as a process for restructuring a current software system to carry out improvements without changing its behavior. The main purpose in refactoring is to improve the quality of a program. This is achieved by reorganizing classes, variables, and methods across the class hierarchy to enable future adaptations and extensions, so that the source code can have better structure, readability, and understandability [18]. In addition to improving the internal structure of the code, refactoring can provide other benefits such as removing duplication of code, improving the design, making the code easier to understand and helping to program faster. There are many refactoring techniques such as composing methods (i.e., extract method, inline method), moving features between objects (i.e., move method), and simplifying method calls (i.e., rename method). The *Extract Method* refactoring enables to move a contiguous code fragment into a separated method, replacing the old code with a call to the new method. To extract a non-contiguous code fragment to a separate method one can use *Program Slicing*, a technique for simplifying programs by focusing on selected aspects of semantics [13].

1.1 Goal of this thesis

There are many slicing algorithms; see, for example, [19]. Most slicing algorithms use program-dependence graph (PDG) as its program representation. In this thesis we use and formalize a different program representation called slide-dependence graph (slideDG), which was introduced by Cozocaru [4]. Cozocaru showed a significant 10%-63% run-time improvement in using slideDG rather then PDG for many algorithms such as slicing, sliding, tucking, bucketing, etc.

In this thesis we present a backward, static, syntax-preserving slicing algorithm, which uses slideDG as its program representation. Our work elaborates on the static single assignment form (SSA)-based slicing algorithm described in the PhD thesis of Ettinger [7]. The algorithm is proved to be semantics-preserving, yet it requires time polynomial in the size of the program. We define a new slicing algorithm that is asymptotically faster and easier to implement than the SSA-based algorithm.

1.2 Background

In what follows, we provide some needed background on slicing.

1.2.1 Program slicing

Program slicing was invented by Mark Weiser [20] as a method for producing a subprogram that preserves a subset of the behavior of the original program. This sub-program is called a *slice*, with respect to a certain *slicing criterion*. A *slicing criterion* is a pair <p, V>, where *p* is a certain point of interest in the program and *V* is a set of variables. For all the examples in this chapter, the point of interest is marked as a statement number in a given program. The slice consists of all the statements in the program (up to the specified point of interest) that directly or indirectly effect the value of the variables given in the slicing criterion. Any slice of a given program preserves the following property: for the initial values of each variable in the program, if the original program terminates, the slice also terminates with the same final values at point *p* of each variable in the program. In this thesis, the point of interest *p* is always the end of a code-fragment of a given program which we choose to slice from, therefore we only use a set of variables as the slicing criterion.

The leading program representation for calculating a slice is PDG, program dependence graph. The PDG represents a program as a graph in which the nodes are statements and predicate expressions and the edges incident to a node represent either the data values on which the node's operations depend and the control conditions on which the execution of the operations depends [9]. In this thesis, we use a variation on the PDG called SlideDG, which will be described later. Note that the original intention of slicing was debugging [20, 21], but since then there have been many other applications to slicing such as software maintenance, testing, program differencing, refactoring [2].

Our first slicing example[1] is given in Listings 1.1-1.3 where the variables *even* and *odd* respectively count the number of even and odd values in a given array of integers. Note that the statements from the original program that are not in the slice are written as an empty statement (blank line).

[1]In this thesis, the programs are written in a deterministic variation on Dijkstra's guarded commands [6], as defined in [7].

```
i := 0;
even := 0;
odd := 0;
while i < a.length do
  if a[i] % 2 == 0 then
    even := even + 1
  else
    odd := odd + 1
  fi;
  i := i + 1
od
```

LISTING 1.1: Original program P1

```
i := 0;
even := 0;

while i < a.length do
  if a[i] % 2 == 0 then
    even := even + 1
  else

  fi;
  i := i + 1
od
```

LISTING 1.2: A slice of P1 with V={even}

```
i := 0;

odd := 0;
while i < a.length do
  if a[i] % 2 == 0 then

  else
    odd := odd + 1
  fi;
  i := i + 1
od
```

LISTING 1.3: A slice of P1 with V={odd}

Static vs. dynamic slicing An important distinction is between static and dynamic slicing. Static slicing computes a slice with no assumptions on the program's input. Dynamic slicing computes a slice for a specific input [14]. In what follows we refer to the program in Listing 1.4.

```
read(n);
i := 1;
result := 1;
while i <= n do
  result := result * i;
  i := i + 1
od;
print(result)
```

LISTING 1.4: Original program P2

In static slicing, all irrelevant statements to the slicing criterion are not included in the slice, as shown in Listing 1.5 regarding the program in Listing 1.4.

```
1  read(n);
2  i := 1;
3  result := 1;
4  while i <= n do
5    result := result * i;
6    i := i + 1
7  od;
8  print(result)
```

LISTING 1.5: A static slice of <8, result> for P2

In dynamic slicing, as shown in Listing 1.6 regarding the program in Listing 1.4, only the relevant statements to the specific input are in the slice.

```
1  read(n);
2  i := 1;
3  result := 1;
4
5
6
7
8  print(result)
```

LISTING 1.6: A dynamic slice for n=0
of <8, result> for P2

Backward vs. forward slicing Another important distinction between two possible directions of slicing is backward and forward slicing. A backward slice of a program and a slicing criterion <p, V> includes all statements and predicates that may have affected the value of the variables in V *up to* the program point p. Backward slicing can be helpful for debugging. A forward slice of a program and a slicing criterion <p, V> contains all the statements and predicates that may be affected by the values of the variables in V *from* the program point p. It can be useful to determine which statements will be affected by changing the value of the variable in the slicing criterion, but it is usually not executable. Examples for both directions are given in Listings 1.7 and 1.8 (both referring to original program P2 in Listing 1.4).

```
1  read(n);
2  i := 1;
3
4  while i <= n do
5
6    i := i + 1
7  od
8
```

LISTING 1.7: Backward slice of <8, i>

```
1
2
3  result := 1;
4
5    result := result * i;
6
7
8  print(result)
```

LISTING 1.8: Forward slice of <3, result>

Syntax-preserving vs. amorphous slicing A syntax-preserving slice is constructed by deleting all statements irrelevant to the slicing criterion from the original program, thus preserving the original program's syntax. By using syntax-preserving slicing we can produce a non-contiguous slice, which we call a *substatement* of the original program. In contrast, an amorphous (semantics-preserving) slice does not have to preserve the program's syntax, and can produce a smaller slice by changing some of the program's statements yet still preserving the original program's behavior [12].

```
1  read(n);
2  if n >= 0 then
3    Skip
4  else
5    n := n * -1
6  fi;
7  print(n)
```

LISTING 1.9: Original program P3

```
1  read(n);
2  if n < 0 then
3    n := n * -1
4  fi;
5  print(n)
```

LISTING 1.10: Amorphous slice of <6, n> for P3

In this thesis we consider syntax-preserving, backward, static slicing algorithm.

1.2.2 Programming notations and representation

Our input program for the slicing algorithm is a compound statement. Each statement in our language is one the following:

- Assignment (LHS, RHS): Two sequences, one for the left-hand side of the assignment and another for the right-hand side of the assignment, used to support multiple assignment.
- Sequential Composition (S1, S2): Two statements executed one after the other.
- If (BoolExp, Then, Else): Boolean expression, a statement for the "then" case and a statement for the "else" case.
- Do (BoolExp, LoopBody): Boolean expression and a statement for the loop body.
- Skip: An empty statement.

Formal declarations for the statements in the language are given in Chapter 2.

1.2.3 Data flow analysis

Data flow analysis is a form of static program analysis and is a process of collecting run-time information about data in programs without executing them [22, 17]. The program representation with data-flow problems is usually the control flow graph (CFG) [3].

Definition 1 (CFG)**.** Control flow graph is a program representation (in the form of a directed graph) of all the possible program's paths during its execution. Each node of the CFG represent a program statement with two additional nodes, entry and exit. Each edge of the CFG (n_1, n_2), represent potential control flow from n_1 to n_2. Normal nodes have one successor, the exit node has no successors, and a predicate node corresponding to a conditional or a loop condition has two successors, where each edge is labeled *True* or *False*. The root of the CFG is the entry node, which is a predicate that has the exit node as its false successor.

An example for a CFG is shown in Figure 1.1.

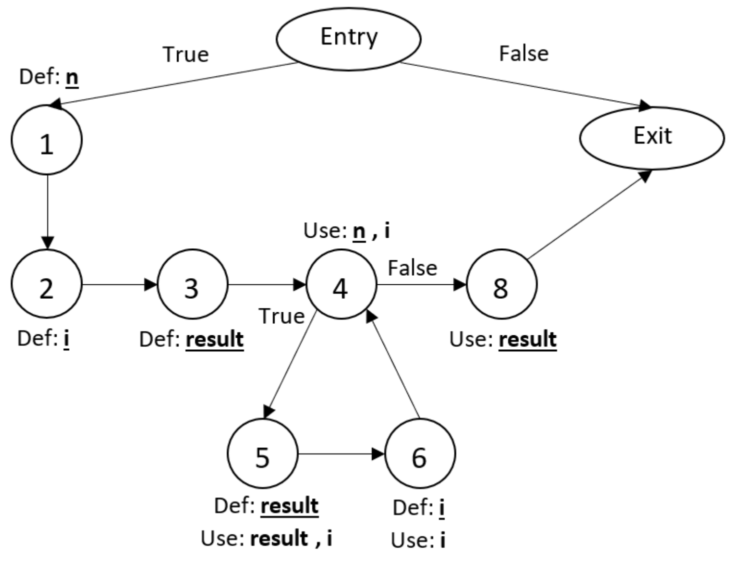

FIGURE 1.1: CFG of P2 (from Listing 1.4). Final-def (Definition 2) and final-use (Definition 3) variable references are underlined.

Definition 2 (Final-def node)**.** *Given a program S, a CFG node n and a variable v, n is considered a final-def node for v iff v is defined in n and there exists a path from n to the exit free of definitions of v.*

Definition 3 (Final-use node)**.** *Given a program S, a CFG node n and a variable v, n is considered a final-use node for v iff v is used in n and each path from n to the exit is free of definitions of v.*

We now describe two classical data-flow analyses: reaching definitions analysis and live-variable analysis.

Reaching definitions analysis

Reaching definitions analysis is a data-flow analysis of computing the set of definitions that *reach* a point in the program (where a definition is a pair of (variable, statement number)). Let v be a variable in the left-hand side of an assignment in a given program point p. This assignment may *reach* a program point p' if v is defined at p, used at p', and there is a CFG path from p to p' free from definitions of v. The set of definitions that reach the exit of a statement S is calculated by: $\text{RD}_{\text{out}}(\text{S}) = (\text{RD}_{\text{in}}(\text{S}) \setminus \text{Kill}(\text{S})) \cup \text{Gen}(\text{S})$, where RD_{in} is the set of definitions at the entry of S, *Kill* is the set of definitions removed by S, and *Gen* is the set of definitions generated by S. Reaching definitions information is usually given by the pair (RD_{in}, RD_{out}) which holds the reaching definitions at the entry of a program and at the end of a program, respectively. In Listing 1.4, $\text{RD}_{\text{out}}(5)$ is calculated using:

- $\text{RD}_{\text{in}}(5)$ = {(n, 1), (i, 2), (result, 3), (result, 5), (i, 6)}
- Kill(5) = {(result, 3), (result, 5)}
- Gen(5) = {(result, 5)}

Therefore: $\text{RD}_{\text{out}}(5)$ = {(n, 1), (i, 2), (result, 5), (i, 6)}, where each definition is given by the pair (variable, statement number).

Live-variable Analysis

Live-variable analysis is another data-flow analysis computed in the direction opposite to the flow of control in a program (backward analysis) [1]. In live-variable analysis, a variable v is considered *live* at the exit of a program point p if there exists a CFG path (that does not define v) from p to a use of v [17]. Otherwise, v is considered *dead* at p. The set of variables live at the entry of a statement S is calculated by: $\mathrm{LV}_{\mathrm{entry}}(\mathrm{S}) = (\mathrm{LV}_{\mathrm{exit}}(\mathrm{S}) \setminus \mathrm{def}(\mathrm{S})) \cup \mathrm{use}(\mathrm{S})$, where LV_{exit} is the set of variables live at the exit of S, *def* is the set of variables defined in S, and *use* is the set of variables whose values may be used in S prior to any definition of the variable. In Listing 1.4, $\mathrm{LV}_{\mathrm{entry}}(5)$ is calculated using:

- $\mathrm{LV}_{\mathrm{exit}}(5) = \{\mathrm{result, i}\}$
- $\mathrm{def}(5) = \{\mathrm{result}\}$
- $\mathrm{use}(5) = \{\mathrm{result, i}\}$

Thus: $\mathrm{LV}_{\mathrm{entry}}(5) = \{\mathrm{result, i}\}$.

In this thesis, we use both reaching definitions and live-variable analyses for the proof of our slicing algorithm.

1.2.4 Slips and slides

Our algorithm represents a statement by a set of slides. Before we show our algorithm, we present two terms that are used in the algorithm by means of an example *slip* and *slide*. Let S be the statement: *"if num > 0 then pos := 1 else pos := 0 fi"*. The slips of S are: *"pos := 1"*, *"pos := 0"*, and *"if num > 0 then pos := 1 else pos := 0 fi"*, while the slides of S are: *"if num > 0 then pos := 1 fi"* and *"if num > 0 then Skip else pos := 0 fi"*.

A slip is any part of a statement which is in itself a statement. A slide is an assignment with all the control-statements (if, do) and sequential-compositions in which it is contained. In case of a multiple-assignment, each assignment (with the relevant control-statements) is considered a slide. In reference to a program representation of a tree, a slip is a subtree and a slide is a path from the root to a certain leaf (described in Chapter 2). For further examples, let us refer back to P1 in Listing 1.1. Some slips there are: *"odd := 0"*, *"i := i + 1"*, and also *"if a[i] % 2 == 0 then even := even + 1 else odd := odd + 1 fi"*; and some slides are given in Listings 1.11-1.13.

```
Skip;
Skip;
odd := 0;
Skip







```

LISTING 1.11: Slide example 1

```
Skip;
Skip;
Skip;
while i < a.length do
    Skip;




  i := i + 1
od
```

LISTING 1.12: Slide example 2

```
Skip;
Skip;
Skip;
while i < a.length do
  if a[i] % 2 == 0 then
    even := even + 1
  else
    Skip;
  fi;
  Skip
od
```

LISTING 1.13: Slide example 3

We also define the union of slides into a statement. Given a statement S and two slides S_m and S_n of S, each program point in the resulting statement contains the statement that is not Skip from the corresponding programs points in S_m and S_n. In the case that both corresponding programs points in S_m and S_n contains Skip, the resulting statement also contains Skip. An example is given in Listing 1.14 for the union of the slides from Listings 1.11 and 1.12.

```
Skip;
Skip;
odd := 0;
while i < a.length do
    Skip;




  i := i + 1
od
```

LISTING 1.14: Union of slides from Listings 1.11 and 1.12

Formal definitions of *slip* and *slide* are given in Chapter 2. In this thesis we use slides as a program representation for our algorithm, and both slips and slides in the proof of our algorithm.

1.2.5 Slide dependence

The algorithm we have defined is slide-based and uses a representation of a program called a slide-dependence graph (slideDG) which was introduced by Cozocaru [4]. For the following definitions we use two terms:

- For a slide s and a variable v we say that v is *defined* in s if v is on the left-hand side of the assignment in s.
- For a slide s and a variable v we say that v is *used* in s if v is on the right-hand side of the assignment in s, or is used in one of the control-statements of s.

Definition 4 (Slide Dependence). *There is a slide dependence due to variable v between two 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 5 (SlideDG)**.** *A Slide-dependence graph (SlideDG) is a program representation (in the form of a directed graph) with slides as nodes, each slide S_n corresponds to a CFG node n, and there is an edge from slide S_n to slide S_m iff S_m is slide-dependent on S_n.*

A formal definition of *slide-dependence* and *slide-dependence graph* is given in Chapter 2, and an example[2] is given in Figure 1.2.

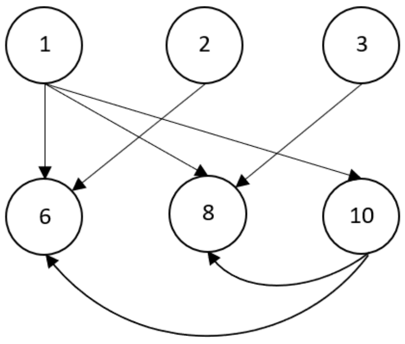

FIGURE 1.2: SlideDG of P1 (from Listing 1.1).

Definition 6 (Final-def Slide)**.** *A slide S_n is called a final-def slide of a statement S and a set of variables V iff the assignment of S_n is to a variable v such that $v \in V$, performed at a CFG node n, and is reaching the exit of S.*

In Figure 1.2 and for V={*even*}, slides 2 and 6 are the final-def slides.

1.2.6 Dafny

The formal framework of this thesis is written in Dafny (1.9.9), a programming language that verifies that the programmer writes correct code with no run-time errors using a verifier. The verifier is used to verify the correctness of the program, while the programmer is writing the code [16]. The verification relies on program specification and annotations such as pre- and post-conditions, loop invariants, assertions, and lemmas.

1.2.7 Static single assignment

The static single assignment (SSA) form, introduced by Cytron et al., is a program representation in which every variable is assigned only once and defined before it is used [5]. Usually, a program is converted to SSA form in order to perform some sort of a program analysis, and then converted back to its original form. In Cytron's approach there are two steps in order to convert a program S and a set of variables V into an SSA form: Inserting assignments called Φ-functions to certain points in the program (control-flow merge points), where the operands to a Φ-function indicate which assignments to a certain variable v reach the merge point. Then, renaming each instance of $v \in V$ to a new name v_i (and increasing i for each instance). In our simple language, instead of using Φ-functions (which are not executable), they can be separated into two assignments. For an IF statement, we use an assignment at the end of each branch, and for a DO statement we use an assignment before the loop

[2]In this thesis, the slide-dependence graph does not include self edges, since their presence has no influence on the results of our algorithm.

and at the end of its body. Examples are given in Listings 1.15, 1.16, and 1.17. The algorithm to converting a program into an SSA form and back from SSA used in this thesis was presented by Ettinger [7] and will be demonstrated next.

```
read($x$);
if $x$ > 0 then
  $y$ := 1
else
  $y$ := 2
fi;
print($y$)
```

LISTING 1.15: A simple program

```
read($x_1$);
if $x_1$ > 0 then
  $y_2$ := 1
else
  $y_3$ := 2
fi;
$y_4$ := $\Phi$($y_2$,$y_3$);
print($y_4$)
```

LISTING 1.16: SSA form

```
read($x_1$);
if $x_1$ > 0 then
  $y_2$ := 1;
  $y_4$ := $y_2$
else
  $y_3$ := 2;
  $y_4$ := $y_3$
fi;
print($y_4$)
```

LISTING 1.17: SSA form used in this thesis

1.2.8 SSA-based slicing

An SSA-based slicing algorithm was presented and proved to be semantics-preserving by Ettinger [7]. The algorithm first converts a statement into SSA form. Then, it computes the flow-insensitive slice on a given set of variables (as explained next). Finally, it returns back from SSA resulting in a flow-sensitive slice. The algorithm has the complexity of O(n^3), where n is the number of statements in the program. We demonstrate the three steps of the algorithm using the example in Listing 1.18.

```
i := 0;
sum := 0;
prod := 1;
while i < a.length do
  sum := sum + a[i];
  prod := prod + a[i];
  i := i + 1
od
```

LISTING 1.18: Original statement S

Translate a statement into SSA form

For the first step, given a statement S and a set of variables V the algorithm translates S into SSA form, while keeping a mapping of each variable and its set of instances. In our example, the statement S' in Listing 1.19 is the SSA form result of the statement S in Listing 1.18.

```
i1 := 0;
sum2 := 0;
prod3 := 1;
i4, sum4, prod4 := i1, sum2, prod3;
while i4 < a.length do
  sum5 := sum4 + a[i4];
  prod6 := prod4 + a[i4];
  i7 := i4 + 1;
  i4, sum4, prod4 := i7, sum5, prod6
od
```

LISTING 1.19: S′ := ToSSA(S, V), where *V*={sum}

Compute the flow-insensitive slice

For the second step, given a statement S and a set of variables V, the algorithm computes the smallest possible slide-independent superset V* of V, and then computes the union of slides of S on V*. In our example, the statement *SV′* in Listing 1.20 is the flow-insensitive slice of the statement *S′* in Listing 1.19 on *V′*, where *V′* consists of the live-on-exit instance of each variable in *V*. Note that in SSA form each variable has at most one live instance in any point in the program.

```
i1 := 0;
sum2 := 0;

i4, sum4 := i1, sum2;
while i4 < a.length do
  sum5 := sum4 + a[i4];

  i7 := i4 + 1;
  i4, sum4 := i7, sum5
od
```

LISTING 1.20: SV′ := ComputeFISlice(S′, V′), where *V′*={sum4}

Translate a statement back from SSA form

For the third and final step, given a statement *S*, the algorithm translates *S* back from SSA form, while using the previous mapping of each variable to its set of instances. In our example, the statement *res* in Listing 1.21 is the translated result of the statement *SV′* in Listing 1.20.

```
i := 0;
sum := 0;

while i < a.length do
  sum := sum + a[i];

  i := i + 1
od
```

LISTING 1.21: res := FromSSA(SV′, V′)

To and from SSA specification

In order to succeed in the proof of correctness of our algorithm, we formulated a formal functional specification for both the To-SSA and From-SSA algorithms, as follows:

```
RemoveEmptyAssignments(Rename(S', XLs, X, globS)) = S
```

where:

- *S*: The original statement.
- *S′*: The SSA version of *S*.
- *X*: A sequence containing all variables defined in *S*.
- *XLs*: A mapping between each variable from *X* to its set of instances.
- *globS*: The set of glob(S) (used to verify that the mapping is valid).
- *Rename*: A function that renames each instance in a given statement to its original variable. In the case where an assignment was renamed into a self-assignment, that assignment is replaced with an empty-assignment.
- *RemoveEmptyAssignments*: A function that removes all empty assignments in a given statement.

However, this specification is bounded by two preconditions:

1. No self assignments: If there is a self assignment in S, for example *x := x*, its SSA version in S′ can be *x1 := x2*. According to our specification, the Rename function renames the assignment back to *x := x* and replaces the self assignment with an empty assignment. Then, the RemoveEmptyAssignments function removes the empty assignment and our specification would not hold.
2. No empty assignments: If there is an empty assignment in S, its SSA version in S′ would still be an empty assignment. According to our specification, the RemoveEmptyAssignments function removes the assignment and our specification would not hold.

For example, *S* is the statement from Listing 1.18, *S′* is the statement from Listing 1.19, *X* is the sequence of variables [i, sum, prod], and *XLs* is the mapping of variables [{i1, i4, i7}, {sum2, sum4, sum5}, {prod3, prod4, prod6}]. When we use Rename on *SV′* from Listing 1.20, we get the statement in Listing 1.22. Then, when we use RemoveEmptyAssignments on the result of Rename on *SV′* we get *res* from Listing 1.21.

```
i := 0;
sum := 0;

[] := [];
while i < a.length do
  sum := sum + a[i];

  i := i + 1;
  [] := []
od
```

LISTING 1.22: Rename(SV′, XLs, X)

Formal definitions of both *Rename* and *RemoveEmptyAssignments* are given in Appendix A.1.4.

1.3 Contributions

The main contributions of this thesis are as follows:

1. It formalizes the definitions of slide-dependence graph and varSlide-dependence graph, and the connection between the two.
2. It provides a functional specification for the transition of a program into SSA, the computation of its flow-insensitive slice, and the transition back from SSA.
3. It proves that any slicing algorithm that meets the requirements listed above is syntax-preserving, in particular the SSA-based algorithm presented in 1.2.8, thus compatible to code-motion refactoring.
4. It provides a new, efficient, semantics- and syntax-preserving slicing algorithm.

Thesis outline The rest of the thesis is structured as follows: Chapter 2 provides a formal framework for this thesis. Chapter 3 elaborates on the correspondence between the definitions in the formal framework. Chapter 4 presents our slide-based slicing algorithm. In Chapter 5 we provide the syntax-preservation proof of our algorithm. Chapter 6 concludes the thesis and suggests ideas for future work.

Chapter 2

Formal framework

In this chapter, we provide the formal framework for our thesis. Here we formally define the slide-dependence graph and varSlide-dependence graph of a program.

2.1 Running example

We illustrate the formal framework of this thesis and the correctness proof of our slicing algorithm (Chapter 5) using the example in Listing 2.1 that calculates the sum of numbers in a given non-empty array, finds its max, and checks whether the array is in a strictly ascending order. The example is an adaptation of an example presented in the VerifyThis competition[1].

```
max := a[0];
sum := a[0];
i := 1;
count := 0;
while i < a.length do
  if a[i] > max then
    max := a[i]
  else
    skip
  fi;
  sum := sum + a[i];
  if max > a[i-1] then
    count := count + 1
  else
    skip
  fi;
  i := i + 1
od;
if count + 1 == a.length then
  isSorted := true
else
  isSorted := false
fi
```

LISTING 2.1: Original program S

[1] https://www.pm.inf.ethz.ch/research/verifythis.html

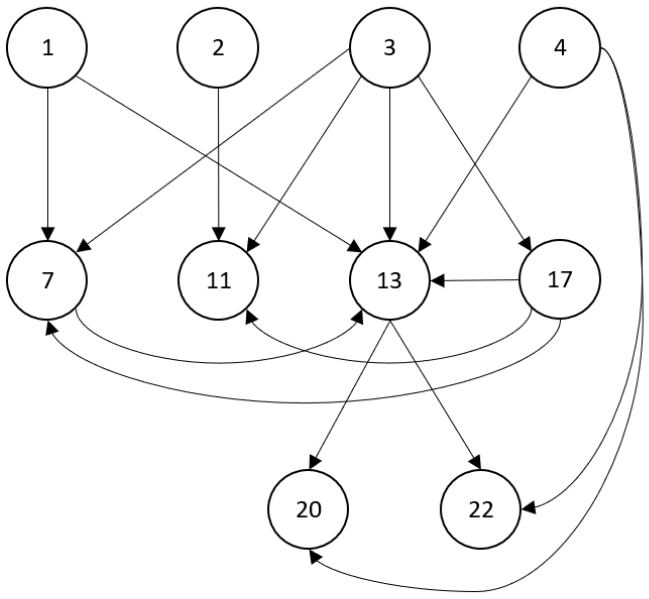

FIGURE 2.1: SlideDG of S from Listing 2.1

```
max1 := a[0];
sum2 := a[0];
i3 := 1;
count4 := 0;
max5, sum5, i5, count5 := max1, sum2, i3, count4;
while i5 < a.length do
  if a[i5] > max5 then
    max7 := a[i5];
    max6 := max7
  else
    max6 := max5
  fi;
  sum8 := sum5 + a[i5];
  if max6 > a[i5-1] then
    count10 := count5 + 1;
    count9 := count10
  else
    count9 := count5
  fi;
  i11 := i5 + 1;
  max5, sum5, i5, count5 := max6, sum8, i11, count9
od;
if count5 + 1 == a.length then
  isSorted13 := true;
  isSorted12 := isSorted13
else
  isSorted14 := false;
  isSorted12 := isSorted14
fi
```

LISTING 2.2: SSA Version of S from Listing 2.1

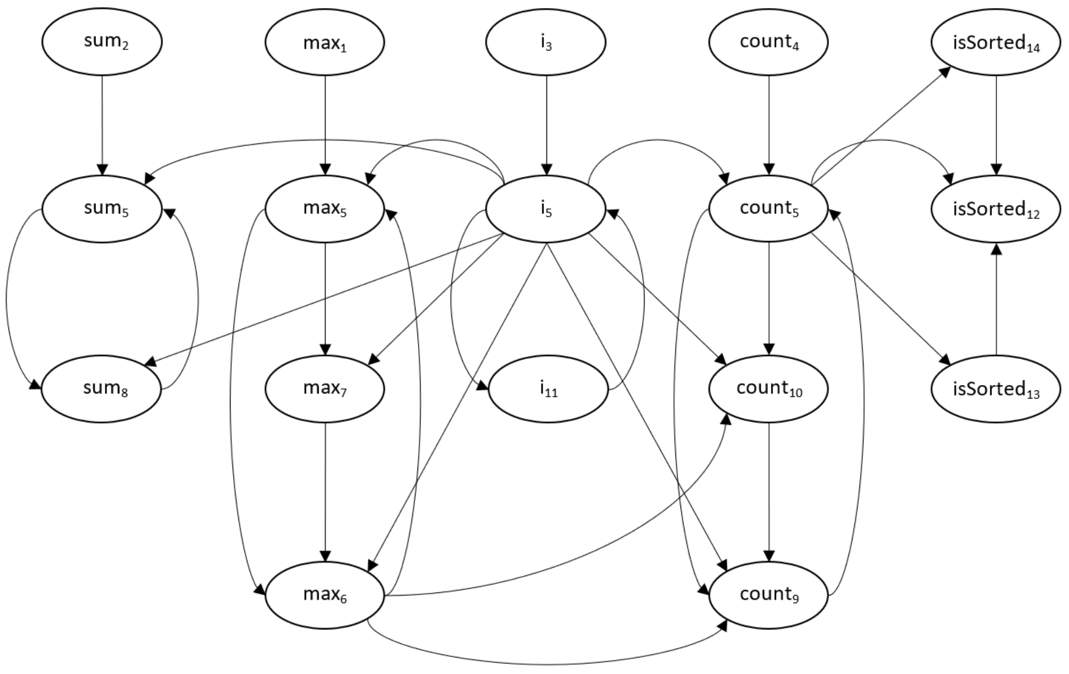

FIGURE 2.2: VarSlideDG of Listing 2.2.

2.2 Programming notations and representation

As mentioned in the previous chapter, our input program is a compound statement. The declarations in Dafny for the statements in the language are as follows:

```
datatype Statement =
    Assignment(LHS: seq<Variable>, RHS: seq<Expression>)
    | SeqComp(S1: Statement, S2: Statement)
    | IF(B0: BooleanExpression, Sthen: Statement, Selse: Statement)
    | DO(B: BooleanExpression, Sloop: Statement)
    | Skip
    | LocalDeclaration(L: seq<Variable>, S0: Statement)
    | Live(L: seq<Variable>, S0: Statement)
    | Assert(B: BooleanExpression)
```

where:

```
type Variable = string
type Expression = (State -> Value, set<Variable>, string)
type BooleanExpression = (State -> bool, set<Variable>)
datatype Value = Int(i: int) | Bool(b: bool)
type State = map<Variable, Value>
```

The input program language for the SSA-based algorithm uses the entire definition of Statement, whereas in this thesis we only use Assignment, SeqComp, IF, DO, and Skip as our core language.

We represent a statement as a tree, where leaves represent assignments or skips and inner nodes represent composite statements. A *label* is a sequence of numbers representing a path from the root to a certain statement in the tree. In Dafny we define a label as a sequence of branches, where a Branch is the number 1 or 2. An example for the representation of a statement as a tree is shown in Figure 2.3. The declaration in Dafny is the following:

```
newtype Branch = b: int | 1 ≤ b ≤ 2
type Label = seq<Branch>
```

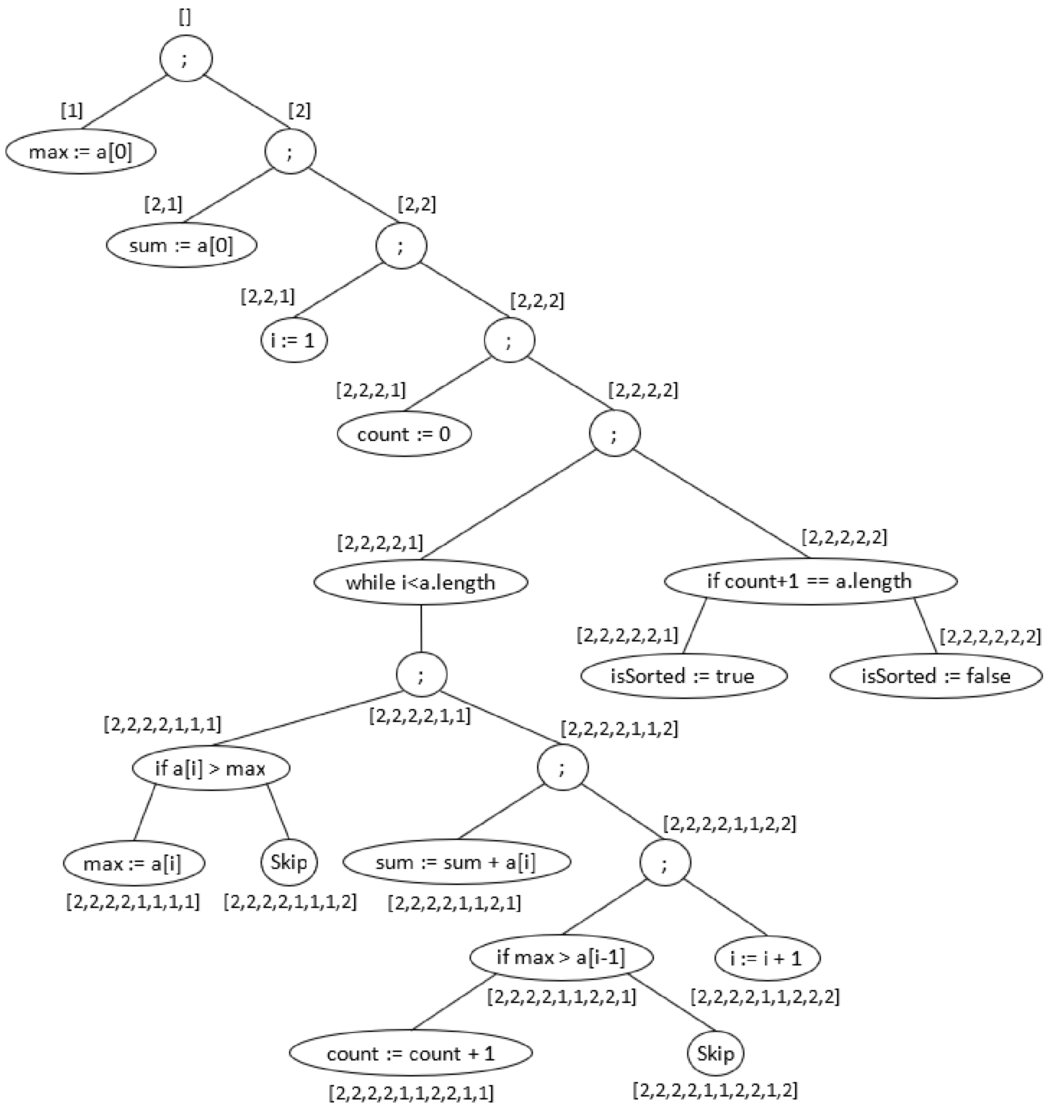

FIGURE 2.3: Representation of the program in Listing 2.1

We can use the definition of *label* when defining a slip of a statement. The formal definition of slip in Dafny is described below. The *slipOf* function uses a number of predicates defined in Appendix A.1.1. *Valid* checks if S is a valid statement (for example, |LHS| = |RHS| if the statement is an assignment); *Core* checks if S is either Assignment, SeqComp, IF, DO, or Skip; and *ValidLabel* checks if l is a valid label in the statement S.

```
function slipOf(S: Statement, l: Label): Statement
  requires Valid(S) ∧ Core(S)
  requires ValidLabel(l, S)
  ensures Valid(slipOf(S, l)) ∧ Core(slipOf(S, l))
  decreases |l|
{
  if l = [] then S
  else
```

```
    match S {
      case SeqComp(S1,S2) ⇒
        if l[0] = 1 then slipOf(S1, l[1..])
        else slipOf(S2, l[1..])
      case IF(B0,Sthen,Selse) ⇒
        if l[0] = 1 then slipOf(Sthen, l[1..])
        else slipOf(Selse, l[1..])
      case DO(B,S1) ⇒
        slipOf(S1, l[1..])
    }
}
```

2.3 Program analysis

In this section we provide our formal definitions of reaching definitions and live-variable analysis presented in Chapter 1.

2.3.1 Reaching definitions

The formal definition of reaching definitions in Dafny is described below. We denote the set of definitions that reach the beginning of a statement as *reaching definitions in*, and the set of definitions that reach the end of a statement as *reaching definitions out*.

Reaching definitions in

```
function ReachingDefinitionsIn(S: Statement, l: Label):
  set<(Variable, Label)>
  requires Valid(S) ∧ Core(S)
  requires ValidLabel(l, S)
{
  ReachingDefinitionsInRec(S, l, S, [], {})
}

function ReachingDefinitionsInRec(slipOfS: Statement,
  l1: Label, S: Statement, l2: Label,
  rdIn: set<(Variable, Label)>): set<(Variable, Label)>
  requires Valid(S) ∧ Core(S)
  requires Valid(slipOfS) ∧ Core(slipOfS)
  requires ValidLabel(l1, slipOfS)
  requires ValidLabel(l2, S)
{
  if l1 = [] then
    if IsDO(slipOfS) then
      rdIn + ReachingDefinitionsOutRec(S, {}, l2+[1])
    else rdIn
  else
    assert ¬IsAssignment(slipOfS) ∧ ¬IsSkip(slipOfS);
    match slipOfS {
    case SeqComp(S1,S2) ⇒
      if l1[0] = 1 then ReachingDefinitionsInRec(S1, l1[1..],
        S, l2+[1], rdIn)
      else ReachingDefinitionsInRec(S2, l1[1..],
        S, l2+[2], ReachingDefinitionsOutRec(S, rdIn, l2+[1]))
    case IF(B0,Sthen,Selse) ⇒
      if l1[0] = 1 then ReachingDefinitionsInRec(Sthen, l1[1..],
        S, l2+[1], rdIn)
      else ReachingDefinitionsInRec(Selse, l1[1..],
        S, l2+[2], rdIn)
    case DO(B,Sloop) ⇒
      ReachingDefinitionsInRec(Sloop, l1[1..],
        S, l2+[1], rdIn + ReachingDefinitionsOutRec(S, {}, l2+[1]))
    }
}

function ReachingDefinitionsInFor(S: Statement, l: Label,
  v: Variable): set<(Variable, Label)>
  requires Valid(S) ∧ Core(S)
  requires ValidLabel(l, S)
{
  var rdIn := ReachingDefinitionsIn(S, l);
  set p | p in rdIn ∧ p.0 = v
}
```

Reaching definitions out

```
function ReachingDefinitionsOut(S: Statement, l: Label):
  set<(Variable, Label)>
  requires Valid(S) ∧ Core(S)
{
  var rdIn := ReachingDefinitionsIn(S, l);
  ReachingDefinitionsOutRec(S, rdIn, l)
}

function ReachingDefinitionsOutRec(S: Statement,
  rdIn: set<(Variable, Label)>, l: Label): set<(Variable, Label)>
  requires Valid(S) ∧ Core(S)
  requires ValidLabel(l, S)
{
  match slipOf(S, l) {
  case Assignment(LHS,RHS) ⇒ RDKill(LHS, rdIn) + RDGen(LHS, l)
  case Skip ⇒ rdIn
  case SeqComp(S1,S2) ⇒
    ReachingDefinitionsOutRec(S,
      ReachingDefinitionsOutRec(S, rdIn, l+[1]), l+[2])
  case IF(B0,Sthen,Selse) ⇒
    ReachingDefinitionsOutRec(S, rdIn, l+[1]) +
    ReachingDefinitionsOutRec(S, rdIn, l+[2])
  case DO(B,Sloop) ⇒
    rdIn + ReachingDefinitionsOutRec(S, {}, l+[1])
  }
}

function ReachingDefinitionsOutFor(S: Statement, l: Label,
  v: Variable): set<(Variable, Label)>
  requires Valid(S) ∧ Core(S)
{
  var rdOut := ReachingDefinitionsOut(S, l);
  set p | p in rdOut ∧ p.0 = v
}
```

Kill and Gen functions

```
function RDKill(V: seq<Variable>, rdIn: set<(Variable, Label)>):
    set<(Variable, Label)>
{
  if V = [] then rdIn
  else
    var s := set p | p in rdIn ∧ V[0] = p.0;
    RDKill(V[1..], rdIn − s)
}

function RDGen(V: seq<Variable>, l: Label): set<(Variable, Label)>
{
  set v | v in V • (v, l)
}
```

For brevity, we denote the *ReachingDefinitionsIn* function by RD_IN(S, l), *ReachingDefinitionsInFor* function by RD_IN_FOR(S, l, v), *ReachingDefinitionsOut* function by RD_OUT(S, l), and *ReachingDefinitionsOutFor* function by RD_OUT_FOR(S, l, v).

For the following examples on program *S* in Listing 2.1, we denote the labels in the sets of RD_IN, RD_IN_FOR, RD_OUT and RD_OUT_FOR as the number of statement in the given program:

- RD_IN(S, [2,2,2,2]) = {(max, 1), (sum, 2), (i, 3), (count, 4)}.
- RD_IN_FOR(S, [2,2,2,2], count) = {(count, 4)}
- RD_OUT(S, []) = {(max, 1), (sum, 2), (i, 3), (count, 4), (max, 7), (sum, 11), (count, 13), (i, 17), (isSorted, 20), (isSorted, 22)}.
- RD_OUT_FOR(S, [], count) = {(count, 4), (count, 13)}.

2.3.2 Liveness

The formal definition of liveness anaylsis in Dafny is described below. We denote the set of variables that are live at the beginning of a statement as *live on entry*, and the set of variables that are live at the end of a statement as *live on exit*.

Live on entry

```
function LiveOnEntry(S: Statement, lvExit: set<Variable>):
  set<Variable>
  requires Valid(S) ∧ Core(S)
{
  match S {
  case Assignment(LHS,RHS) ⇒ LVKill(LHS, liveOnExit) + LVGen(RHS)
  case Skip ⇒ lvExit
  case SeqComp(S1,S2) ⇒ LiveOnEntry(S1, LiveOnEntry(S2, lvExit))
  case IF(B0,Sthen,Selse) ⇒ LiveOnEntry(Sthen, lvExit) +
    LiveOnEntry(Selse, lvExit) + B0.1
  case DO(B,Sloop) ⇒ lvExit + LiveOnEntry(Sloop, {}) + B.1
  }
}
```

Kill and Gen functions

```
function LVKill(V: seq<Variable>, lvExit: set<Variable>):
  set<Variable>
```

```
{
  if V = [] then lvExit
  else
    var s := set v | v in lvExit ∧ V[0] = v;
    LVKill(V[1..], lvExit − s)
}

function LVGen(E: seq<Expression>): set<Variable>
{
  GetRHSVariables(E)
}
```

The function *GetRHSVariables* can be found in Appendix A.1.1.

Live on exit

```
function LiveOnExit(S: Statement, lvExit: set<Variable>,
  l: Label): set<Variable>
  requires Valid(S) ∧ Core(S)
  requires ValidLabel(l, S)
{
  if l = [] then lvExit else
  assert ¬IsAssignment(S) ∧ ¬IsSkip(S);
  match S {
  case SeqComp(S1,S2) ⇒
    if l[0] = 1 then
      LiveOnExit(S1, LiveOnEntry(S2, lvExit), l[1..])
    else
      LiveOnExit(S2, lvExit, l[1..])
  case IF(B0,Sthen,Selse) ⇒
    if l[0] = 1 then
      LiveOnExit(Sthen, lvExit, l[1..])
    else
      LiveOnExit(Selse, lvExit, l[1..])
  case DO(B,Sloop) ⇒
    LiveOnExit(Sloop, lvExit +
      LiveOnEntry(Sloop, lvExit), l[1..])
  }
}
```

For brevity, we denote the *LiveOnEntry* function by LV_ENTRY(S, lvExit) and *LiveOnExit* function by LV_EXIT(S, lvExit, l).

For example:

- LV_ENTRY(S, {max5, sum5, i5, count5, isSorted12}) = {}.
- LV_EXIT(S, {max5, sum5, i5, count5, isSorted12}, []) = {max5, sum5, i5, count5, isSorted12}.
- LV_EXIT(S, {max5, sum5, i5, count5, isSorted12}, [2,2,2,2,1]) = {max5, sum5, i5, count5}.

2.4 Slides

We recall the definition of a slide from Chapter 1 as an assignment with all the control-statements (if, do) and sequential-compositions in which it is contained. The formal definition of *Slide* is as follows, where *Label* is the label of the assignment in the slide, and *Variable* is the variable defined in the slide. For that we use two accessors, *SlideLabel* and *SlideVariable*, which can be found in Appendix A.1.1.

```
type Slide = (Label, Variable)
```

2.4.1 Slide dependence graph

A formal definition of slide dependence in Dafny is described below. The *SlideDependence* predicate uses *SlidesOf* which returns all the slides of a statement; *def* which returns the set of variables defined in a specific statement; *SlideLabels* which returns a set of the slide's label and all the labels of the control-statements (if, do) and sequential-compositions in which it is contained; *UsedVars* which returns all the variables used for a specific label and statement; and *ReachingDefinition* which checks if a certain pair of (variable, label1) is in RD_IN(S, label2). Full definitions can be found in Appendix A.1.2.

```
predicate SlideDependence(Sm: Slide, Sn: Slide, S: Statement)
  requires Valid(S) ∧ Core(S)
  requires Sm in SlidesOf(S, def(S)) ∧ Sn in SlidesOf(S, def(S))
{
  var v := SlideVariable(Sm);
  ∃ l • l in SlideLabels(Sn, S) ∧
  v in UsedVars(S, l) ∧ ReachingDefinition(S, SlideLabel(Sm), l, v)
}

predicate ReachingDefinition(S: Statement, l1: Label, l2: Label,
  v: Variable)
  requires Valid(S) ∧ Core(S)
  requires ValidLabel(l1, S) ∧ ValidLabel(l2, S)
{
  (v, l1) in ReachingDefinitionsIn(S, l2)
}
```

For example, the variable *max* is defined in slide 1, used in the label *l* = [2,2,2,2,1,1,1,1] which is one of the labels of slide 7, and (*max*, [1]) is in RD_IN(S, *l*), therefore slide 7 is slide-dependent on slide 1.

The formal definition of a *slide-dependence graph* is as follows, where *Statement* is the statement represented by the graph; *set<Slide>* is the set of slides of the graph (nodes); and *map<Slide, set<Slide>>* is the mapping between each slide to its predecessors (edges). For that we use three accessors: *SlideDGStatement*, *SlideDGSlides* and *SlideDGMap*, which can be found in Appendix A.1.2.

```
type SlideDG = (Statement, set<Slide>, map<Slide,set<Slide>>)

function SlideDGOf(S: Statement): SlideDG
  requires Valid(S) ∧ Core(S)
{
  var slides := SlidesOf(S, def(S));
  var m := map s | s in slides •
    SlideDependencePredecessorsOf(s, S);

  (S, slides, m)
}

function SlideDependencePredecessorsOf(Sn: Slide, S: Statement):
  set<Slide>
  requires Valid(S) ∧ Core(S)
  requires Sn in SlidesOf(S, def(S))
{
  set Sm | Sm in SlidesOf(S, def(S)) ∧ SlideDependence(Sm, Sn, S)
}
```

2.4.2 Paths in a slide dependence graph

We describe a path in a slideDG between two slides as *SlideDGPath* with the following definition:

```
datatype SlideDGPath = Empty | Extend(SlideDGPath, Slide)
```

Then, we describe the reachability between two slides as *SlideDGReachable* with the following definition:

```
predicate SlideDGReachable(slideDG: SlideDG, from: Slide,
  to: Slide, slides: set<Slide>)
{
  ∃ via: SlideDGPath •
    SlideDGReachableVia(slideDG, from, via, to, slides)
}

predicate SlideDGReachableVia(slideDG: SlideDG, from: Slide,
  via: SlideDGPath,
  to: Slide, slides: set<Slide>)
  decreases via
{
  match via
  case Empty ⇒ from = to
  case Extend(prefix, n) ⇒ n in slides ∧
    to in SlideDGPredecessors(slideDG, n) ∧
    SlideDGReachableVia(slideDG, from, prefix, n, slides)
}

function SlideDGPredecessors(slideDG: SlideDG, n: Slide):
  set<Slide>
  requires n in SlideDGMap(slideDG)
{
  SlideDGMap(slideDG)[n]
}
```

Another important distinction we make is between *finalDefSlides* and the rest. Given a statement *S* and its slide-dependence graph *slideDG*, it returns the set of slides that are reaching to the exit of *S*.

```
function FinalDefSlides(S: Statement, V: set<Variable>): set<Slide>
  requires Valid(S) ∧ Core(S)
{
  var slideDG := SlideDGOf(S);
  set slide | slide in SlideDGSlides(slideDG) ∧
  SlideVariable(slide) in V ∧
  slide in FinalDefSlidesOfVariable(S, SlideVariable(slide))
}

function FinalDefSlidesOfVariable(S: Statement, v: Variable):
  set<Slide>
  requires Valid(S) ∧ Core(S)
{
  var slideDG := SlideDGOf(S);
  var rdIn := set v | v in def(S) • (v, []);
  var rdOutv := set pair |
    pair in ReachingDefinitionsOutRec(S, rdIn, []) ∧ pair.0 = v;
  var slidesRdOutv := set pair | pair in rdOutv •
    (pair.1, pair.0);
  set slide | slide in slidesRdOutv * SlideDGSlides(slideDG)
}
```

For example, FinalDefSlides(*S*, {*isSorted*}) is the set of slides {20, 22}.

We derive the following definition from *reaching definitions* and *final-def slides*:

Definition 7. *The set of slides RD_OUT(S, []) is the set of all the final-def slides of S.*

2.5 VarSlides

The program representation we use for the SSA form of a program is the *VarSlide-Dependence graph*. Similar to a *Slide-Dependence graph*, the *VarSlide-Dependence graph* uses *VarSlide*s as nodes. The formal definition of *VarSlide* is as follows, where *Variable* is the variable defined in the varSlide, and *VarSlideTag* is the tag of the varSlide, regular when *Variable* is defined only once and phi when *Variable* is defined twice (control-flow merge point). For that we use two accessors, *VarSlideVariable* and *VarSlideTag*, which can be found in Appendix A.1.3.

```
datatype VarSlideTag = Phi | Regular
type VarSlide = (Variable, VarSlideTag)
```

Note that a varSlide does not contain a label, therefore if a varSlide is regular it has only one valid label in a given statement, and if a varSlide is phi it has two valid labels in the statement.

2.5.1 VarSlide dependence graph

The definitions of *varSlide dependence* and *varSlide-dependence graph* are as follows:

Definition 8 (VarSlide Dependence)**.** *There is a varSlide dependence between two varSlides S_m and S_n iff there is a variable v defined in S_m and used in S_n.*

Definition 9 (VarSlideDG)**.** *A VarSlide-dependence graph (VarSlideDG) is a program representation (in the form of a directed graph) with varSlides as nodes and there is an edge between varSlides S_n and S_m iff S_m is varSlide-dependent on S_n.*

A formal definition of varSlide dependence in Dafny is described below. The *VarSlideDependence* predicate uses *VarSlideLabels* which returns a set of the varSlide's label and all the labels of the control-statements (if, do) and sequential-compositions in which it is contained. Full definition can be found in Appendix A.1.3.

```
predicate VarSlideDependence(Sm: VarSlide, Sn: VarSlide,
  S: Statement)
  requires Valid(S) ∧ Core(S)
{
  var v := VarSlideVariable(Sm);
  ∃ l • l in VarSlideLabels(Sn, S) ∧ v in UsedVars(S, l)
}
```

For example, the variable *max5* is defined in the varSlide of *max5* and used in the label *l* = [2,2,2,2,1,2,1,1,1,1,1] of the varSlide of *max7*, therefore the varSlide of *max7* is varSlide-dependent on the varSlide of *max5*.

The formal definition of *varSlide-dependence graph* is as follows, where *Statement* is the statement represented by the graph, *set<VarSlide>* is the set of varSlides of the graph (nodes), and *map<VarSlide, set<VarSlide>>* is the mapping between each varSlide to its predecessors (edges). For that we use three accessors, *VarSlideDGStatement*, *VarSlideDGVarSlides* and *VarSlideDGMap*, which can be found in Appendix A.1.3. The *VarSlideDGOf* function uses *VarSlidesOf* which returns all the varSlides of a statement, and can also be found in Appendix A.1.3.

```
type VarSlideDG = (Statement, set<VarSlide>, map<VarSlide,
  set<VarSlide>>)

function VarSlideDGOf(T: Statement): VarSlideDG
  requires Valid(T) ∧ Core(T)
{
  var varSlides := VarSlidesOf(T, def(T));
  var m := map s | s in varSlides •
    VarSlideDependencePredecessorsOf(s, T);

  (T, varSlides, m)
}

function VarSlideDependencePredecessorsOf(Sn: VarSlide,
  T: Statement): set<VarSlide>
  requires Valid(T) ∧ Core(T)
  requires Sn in VarSlidesOf(T, def(T))
{
  set Sm | Sm in VarSlidesOf(T, def(T)) ∧
    VarSlideDependence(Sm, Sn, T)
}
```

2.5.2 Paths in a varSlide dependence graph

We express the existence of a path in varSlideDG between two varSlides as *VarSlideDGPath* with the following definition:

```
datatype VarSlideDGPath = Empty | Extend(VarSlideDGPath, VarSlide)
```

Then, we describe the reachability between two varSlides as *VarSlideDGReachable* or *VarSlideDGReachablePhi* with the following definitions:

```
predicate VarSlideDGReachable(varSlideDG: VarSlideDG,
  from: VarSlide, to: VarSlide, S: set<VarSlide>)
{
  ∃ via: VarSlideDGPath •
    VarSlideDGReachableVia(varSlideDG, from, via, to, S)
}

predicate VarSlideDGReachableVia(varSlideDG: VarSlideDG,
  from: VarSlide, via: VarSlideDGPath, to: VarSlide,
  S: set<VarSlide>)
  decreases via
{
  match via
  case Empty ⇒ from = to
  case Extend(prefix, n) ⇒ n in S ∧
  to in VarSlideDGNeighbours(varSlideDG, n) ∧
  VarSlideDGReachableVia(varSlideDG, from, prefix, n, S)
}
```

```
function VarSlideDGNeighbours(varSlideDG: VarSlideDG, n: VarSlide):
  set<VarSlide>
  requires n in VarSlideDGMap(varSlideDG)
{
  VarSlideDGMap(varSlideDG)[n]
}

predicate VarSlideDGReachablePhi(varSlideDG: VarSlideDG,
  from: VarSlide, to: VarSlide, S: set<VarSlide>)
{
  ∃ via: VarSlideDGPath • VarSlideDGReachableViaPhi(varSlideDG,
    from, via, to, S)
}

predicate VarSlideDGReachableViaPhi(varSlideDG: VarSlideDG,
  from: VarSlide, via: VarSlideDGPath, to: VarSlide,
  S: set<VarSlide>)
  decreases via
{
  match via
  case Empty ⇒ from = to
  case Extend(prefix, n) ⇒ n in S ∧
  to in VarSlideDGNeighbours(varSlideDG, n) ∧
  n.1 = Phi ∧
  VarSlideDGReachableVia(varSlideDG, from, prefix, n, S)
}
```

2.6 Correspondence between slideDG and varSlideDG

Given a statement S and its corresponding SSA version S', we describe the connection between a slide in the slideDG of S and its corresponding varSlide in the varSlideDG of S' using the following function:

```
function VarSlideOf(S: Statement, S': Statement, slide: Slide,
  XLs: seq<set<Variable>>, X: seq<Variable>): VarSlide
  requires Valid(S) ∧ Valid(S')
  requires Core(S) ∧ Core(S')
  requires ValidXLs(glob(S'), XLs, X)
  requires S = RemoveEmptyAssignments(Rename(S', XLs, X, glob(S)))
  requires slide in SlideDGSlides(SlideDGOf(S))
{
  var v, l := SlideVariable(slide), SlideLabel(slide);
  var l' := VarLabelOf(S, S', l, XLs, X);
  var v' := InstanceOf(S', l', v, XLs, X);
  (v', Regular)
}
```

We first find the variable v and label l of the given slide. Then, we find l's corresponding varLabel l' (using the *VarLabelOf* function), and finally we find the instance i (of variable v) defined in l' (using the *InstanceOf* function). Every slide in the slideDG of S has exactly one corresponding regular varSlide in the varSlideDG of S' (and vice versa), therefore *VarSlideOf* is considered an invertible function. The declarations of functions used in *VarSlideOf* can be found in Appendix A.1.4.

For example, given slide 7:

- The label of slide 7 is l = [2,2,2,2,1,1,1,1].
- The variable of slide 7 is v = *max*.
- The varLabel of l is l′ = [2,2,2,2,1,2,1,1,1,1,1].
- The instance of v defined at l′ is *max7*.
- Therefore, the varSlide corresponding to slide 7 for max is (*max7*, Regular).

Further correspondences between the two graphs are described in the next chapter.

Chapter 3

Properties of slide dependence graphs

In this chapter we provide a deeper look at the correspondence between slideDG and varSlideDG of a program.

3.1 Reaching definitions and liveness

We present two theorems that describe the correspondence between reaching definitions and liveness analysis. Both theorems use the following definition (regular frontier):

$$RF(v') = \begin{cases} \{\} & \text{v}' \text{ is not in def(S)} \\ \{\text{varSlide of v}'\} & \text{v}' \text{ is a regular instance} \\ \text{regular predecessors of the varSlide of v}' & \text{v}' \text{ is a phi instance} \end{cases}$$

Note that for a given varSlide *vSlide*, for each predecessor p of *vSlide*: if p is a regular varSlide, then it is in the *regular predecessors* of *vSlide*; if p is a phi varSlide, then the *regular predecessors* of p are in the *regular predecessors* of *vSlide*.

Theorem 1 (Reaching definitions and liveness for exit)**.** Let S be a statement, and S' be its SSA form. Let l', v' be a valid label in S' and a live on exit instance of S' at l', respectively. Let v, l be the variable of v' and the corresponding label of l' using the *VarLabelOf* function, respectively.
We then say that all varSlides of RD_OUT_FOR(S, l, v) = RF(v').

Examples

Let us demonstrate this theorem on Listings 2.1 and 2.2 using the following examples:

1. For the first example, let l', v' be 15 and *sum8*, respectively, as *sum8* is a live on exit instance of S' at 15. Let v, l be *sum* and 13, respectively, as *sum* is the variable of *sum8* and 13 is the label of 15 in S. The set RD_OUR_FOR(S, 13, *sum*) is {(*sum*, 11)}, and its corresponding set of varSlides is {*sum8*}. The varSlide of *sum8* is Regular, therefore RF(*sum8*) is {*sum8*}.

2. For the second example, let l', v' be 13 and *max6*, respectively, as *max6* is a live on exit instance of S' at 13. Let v, l be *max* and 11, respectively, as *max* is the variable of *max6* and 11 is the label of 13 in S. The set RD_OUR_FOR(S, 11, *max*)

is {(*max*,1), (*max*,7)}, and its corresponding set of varSlides is {*max1*, *max7*}. The varSlide of *max6* is Phi, therefore RF(*max6*) is {*max1*, *max7*}.

3. For the third and final example, let *l′*, *v′* be the empty label [] and *isSorted12*, respectively, as *isSorted12* is a live on exit instance of *S′* at [], meaning its a live on exit instance of the entire statement *S′*. Let *v*, *l* be *isSorted* and [], respectively, as *isSorted* is the variable of *isSorted12* and [] is the label of [] in *S*. The set RD_OUR_FOR(*S*, [], *isSorted*) is {(*isSorted*,20), (*isSorted*,22)} (the final-def slides of *isSorted* in *S* according to Definition 7), and its corresponding set of varSlides is {*isSorted13*, *isSorted14*}. The varSlide of *isSorted12* is Phi, therefore RF(*isSorted12*) is {*isSorted13*, *isSorted14*}.

The exact details of the proof are not immediately needed for understanding the theorem, therefore the formal proof can be found in Appendix A.2.1. However, here we provide a short description of the proof.

In the case that slipOf(*S*, *l*) is an assignment and *v* is defined in that assignment, the set RD_OUT_FOR(*S*, *l*, *v*) includes only the slide of (*v*, *l*). The label *l′* is the var-label of *l*, therefore cannot be a label of a Phi assignment. Thus, the var-slide of (*v*, *l*) is Regular, therefore RF(*v′*) includes only the var-slide itself. Then, all varSlides of RD_OUT_FOR(*S*, *l*, *v*) = RF(*v′*).

In the case that slipOf(*S*, *l*) is an If statement and *v* is defined in that statement, *v′* is an instance of a Phi var-slide. Then, RF(*v′*) is the sum of RF(*v1′*) + RF(*v2′*) where *v1′* and *v2′* are the instances of *v* defined in both branches of the If statement. Inductively, RF(*S′*, *v1′*) = the varSlides of RD_OUT_FOR(*S*, *l*+[1], *v*) and RF(*S′*, *v2′*) = the varSlides of RD_OUT_FOR(*S*, *l*+[2], *v*), where *l*+[1] and *l*+[2] are the labels of both branches of the If statement. The set of RD_OUT for an If statement is the sum of the set of RD_OUT for the Then and the Else branch of the If statement (by definition of RD_OUT in Listing 2.3.1). Therefore the varSlides of RD_OUT_FOR(*S*, *l*+[1], *v*) + the varSlides of RD_OUT_FOR(*S*, *l*+[2], *v*) is exactly the varSlides of RD_OUT_FOR(*S*, *l*, *v*).

Theorem 2 (Reaching definitions and liveness for entry)**.** Let *S* be a statement, and *S′* be its SSA form. Let *l′*, *v′* be a valid label in *S′* and a live on entry instance of *S′* at *l′*, respectively. Let *v*, *l* be the variable of *v′* and the corresponding label of *l′* using the *VarLabelOf* function, respectively.
We then say that all varSlides of RD_IN_FOR(*S*, *l*, *v*) = RF(*v′*).

Examples

The following examples demonstrate the theorem:

1. For the first example, let *l′*, *v′* be 13 and *sum5*, respectively, as *sum5* is a live on exit instance of *S′* at 13. Let *v*, *l* be *sum* and 11, respectively, as *sum* is the variable of *sum5* and 11 is the label of 13 in *S*. The set RD_OUR_FOR(*S*, 11, *sum*) is {(*sum*, 2), (*sum*, 11)}, and its corresponding set of varSlides is {*sum2*, *sum8*}. The varSlide of *sum5* is Phi, therefore RF(*sum5*) is {*sum2*, *sum8*}.

2. For the second and final example, let *l′*, *v′* be 14 and *max6*, respectively, as *max6* is a live on exit instance of *S′* at 14. Let *v*, *l* be *max* and 12, respectively, as *max* is the variable of *max6* and 12 is the label of 14 in *S*. The set RD_OUR_FOR(*S*, 12, *max*) is {(*max*, 1), (*max*, 7)}, and its corresponding set of varSlides is {*max1*, *max7*}. The varSlide of *max6* is Phi, therefore RF(*max6*) is {*max1*, *max7*}.

The formal proof can be found in Appendix A.2.2. However, here we provide a short description of the proof.

Let *l1* be a valid label in *S* (where *l* = *l1* + [c]), *l1′* be the varLabel of *l1*, and slipOf(*S*, *l1*) be a SeqComp statement. In the case that c = 1, the set of LV_ENTRY for the left branch of a SeqComp statement is equal to the set of LV_ENTRY of the SeqComp statement itself (by definition of LV_ENTRY in Listing 2.3.2), therefore *v′* is live on entry at *l1′*. Inductively, RF(*S′*, *v′*) = the varSlides of RD_IN_FOR(*S*, *l1*). The set of RD_IN for a SeqComp statement is equal to the set of RD_IN for the left branch of a SeqComp statement (by definition of RD_IN in Listing 2.3.1), therefore the varSlides of RD_IN_FOR(*S*, *l1*) = the varSlides of RD_IN_FOR(*S*, *l*).

In the case that c = 2, the set of LV_ENTRY for the right branch of a SeqComp statement is equal to the set of LV_EXIT for the left branch of the SeqComp statement (by definition of LV_ENTRY and LV_EXIT in Listing 2.3.2), therefore *v′* is live on exit at *l1′*+[1]. Using Theorem 1, RF(*S′*, *v′*) = the varSlides of RD_OUT_FOR(*S*, *l1*+[1]). The set of RD_OUT for the left branch of a SeqComp statement is equal to the set of RD_IN for the right branch of the SeqComp statement (by definition of RD_IN and RD_OUT in Listing 2.3.1), therefore the varSlides of RD_OUT_FOR(*S*, *l1*+[1]) = the varSlides of RD_IN_FOR(*S*, *l*).

3.2 Reachability in the slideDG and the varSlideDG

We now turn to describe the correspondence between slides and paths of the slide-dependence graph of *S*, and varSlides and paths of the varSlide-dependence graph of *S′*, using the following two theorems:

Theorem 3 (Path correspondence)**.** Given a statement *S*, its slide-dependence graph *slideDG*, its varSlide-dependence graph *varSlideDG*, and two slides *slide1* and *slide2* where *slide2* is reachable from *slide1* in *slideDG* using *via*, then the varSlide of *slide2* is reachable from the varSlide of *slide1* in *varSlideDG*.

Theorem 4 (Path back correspondence)**.** Given a statement *S*, its slide-dependence graph *slideDG*, its varSlide-dependence graph *varSlideDG*, and two slides *slide1* and *slide2* where the varSlide of *slide2* is reachable from the varSlide of *slide1* in *varSlideDG*, then *slide2* is reachable from *slide1* in *slideDG*.

The formal definition and proof of *Path correspondence* is as follows:

```
lemma PathCorrespondence(slideDG: SlideDG, varSlideDG: VarSlideDG,
slide1: Slide, slide2: Slide, via: SlideDGPath, S: Statement,
S′: Statement, XLs: seq<set<Variable>>, X: seq<Variable>)
  requires Valid(S) ∧ Core(S)
  requires Valid(S′) ∧ Core(S′)
  requires IsSlideDGOf(slideDG, S)
  requires IsVarSlideDGOf(varSlideDG, S)
  requires ValidXLs(glob(S), XLs, X)
  requires RemoveEmptyAssignments(Rename(S′, XLs, X, glob(S))) = S
  requires SlideDGReachableVia(slideDG, slide1, via,
    slide2, SlideDGSlides(slideDG))
  ensures VarSlideDGReachable(varSlideDG,
    VarSlideOf(S, S′, slide1, XLs, x),
    VarSlideOf(S, S′, slide2, XLs, x),
    VarSlideDGVarSlides(varSlideDG))
  decreases via
```

Proof. Let $slide1$ and $slide2$ be slides of S, and $varSlide1$ and $varSlide2$ be their corresponding varSlides of S′. We need to show that if $slide2$ is reachable from $slide1$ then $varSlide2$ is reachable from $varSlide1$. Let via be the SlideDGPath between $slide1$ and $slide2$.

Base case: If via is Empty then in fact $slide1 = slide2$, therefore $varSlide1 = varSlide2$ which means that $varSlide2$ is reachable from $varSlide1$.

via is Extend(prefix, n), then n is reachable from $slide1$ using the path $prefix$, and n is a predecessor of $slide2$. Let n' be the corresponding varSlide of n. We need to show that n' is reachable from $varSlide1$ and that $varSlide2$ is reachable from n'.

Inductive hypothesis: Suppose n' is reachable from $varSlide1$.

Inductive step: n is a predecessor of $slide2$, therefore $varSlide2$ is reachable from n' using Lemma 5 (as explained next). We have shown a path between $varSlide1$ and n' and between n' and $varSlide2$, therefore there is a path between $varSlide1$ and $varSlide2$ and we can say that $varslide2$ is reachable from $varSlide1$. □

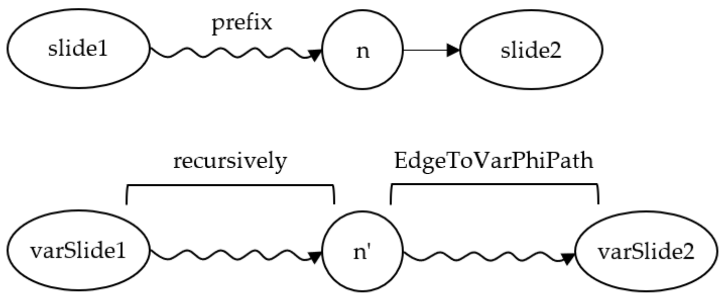

FIGURE 3.1: PathCorrespondence demonstration.

Lemma 5 (Edge to var phi path)**.**

```
lemma EdgeToVarPhiPath(slide1: Slide, slide2: Slide,
slideDG: SlideDG, varSlideDG: VarSlideDG, S: Statement,
S′: Statement, XLs: seq<set<Variable>>, X: seq<Variable>)
  requires Valid(S) ∧ Core(S)
  requires Valid(S′) ∧ Core(S′)
  requires IsSlideDGOf(slideDG, S)
  requires IsVarSlideDGOf(varSlideDG, S)
  requires SlideDependence(slide1, slide2, S)
  requires ValidXLs(glob(S), XLs, X)
  requires RemoveEmptyAssignments(Rename(S′, XLs, X, glob(S))) = S
  ensures VarSlideDGReachablePhi(varSlideDG,
    VarSlideOf(S, S′, slide1, XLs, x),
    VarSlideOf(S, S′, slide2, XLs, x),
    VarSlideDGVarSlides(varSlideDG))
```

Proof. Let v be the variable defined in *slide1* and used in *slide2*. Let *l1* be the label of *slide1* where v is defined, and *l2* be the label of *slide2* where v is used. Let *vSlide1*, *varL1* and *vSlide2*, *varL2* be the corresponding var-slides and labels of *slide1*, *l1* and *slide2*, *l2*, respectively. Let v'' be the instance of v defined in *vSlide1* and v' be the instance of v used in *vSlide2*, which is live on entry in *varL2*.

Using Theorem 2, we can say that all varSlides of RD_IN_FOR(S, *l2*, v) = RF(v'). There is a slide-dependence between *slide1* and *slide2*, therefore (v, *l1*) is in RD_IN_FOR(S, *l2*, v) and its varSlide of *vSlide1* is in RF(v').

If v' is Regular, then by definition RF(v') is {varSlide of v'}. We already saw that *vSlide1* is in RF(v'), therefore the varSlide of v' is exactly *vSlide1* which states that $v'' = v'$. Therefore, there is an edge between *vSlide1* and *vSlide2* in varSlideDG and *vSlide2* is reachable from *vSlide1*.

If v' is Phi, then by definition RF(v') is the regular predecessors of v'. We already saw that *vSlide1* is in RF(v'), therefore there is a path of 0 or more phi varSlides between the *vSlide1* and varSlide of v'. Furthermore, since v' is used in *vSlide2*, there is an edge between the varSlide of v' and *vSlide2*. Therefore, there is a path of 0 or more phi varSlides between *vSlide1* and *vSlide2*, and we can say that *vSlide2* is reachable from *vSlide1*. □

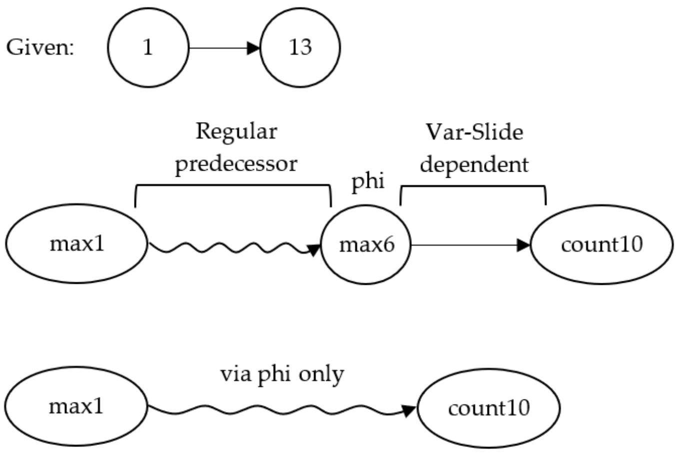

FIGURE 3.2: EdgeToVarPhiPath example: Slide 1 as *slide1*, slide 13 as *slide2*, max1 as v'' and max6 as v'.

The formal definition and proof of *Path back correspondence* is as following:

```
lemma PathBackCorrespondence(slideDG: SlideDG,
varSlideDG: VarSlideDG, slide1: Slide, slide2: Slide,
via: VarSlideDGPath, S: Statement, S': Statement,
XLs: seq<set<Variable>>, X: seq<Variable>)
  requires Valid(S) ∧ Core(S)
  requires Valid(S') ∧ Core(S')
  requires IsSlideDGOf(slideDG, S)
  requires IsVarSlideDGOf(varSlideDG, S)
  requires ValidXLs(glob(S), XLs, X)
  requires RemoveEmptyAssignments(Rename(S', XLs, X, glob(S))) = S
  requires VarSlideTag(VarSlideOf(S, S', slide1, XLs, x)) = Regular
    ∧ VarSlideTag(VarSlideOf(S, S', slide2, XLs, x)) = Regular
  requires VarSlideDGReachableVia(varSlideDG,
    VarSlideOf(S, S', slide1, XLs, x), via,
    VarSlideOf(S, S', slide2, XLs, x),
    VarSlideDGVarSlides(varSlideDG))
  ensures SlideDGReachable(slideDG, slide1, slide2,
    SlideDGSlides(slideDG))
  decreases via
```

Proof. Let *slide*1 and *slide*2 be slides of S, and *varSlide*1 and *varSlide*2 be their corresponding varslides of S′. We need to show that if *varSlide*2 is reachable from *varSlide*1 then *slide*2 is reachable from *slide*1. Let *via* be the VarSlideDGPath between *varSlide*1 and *varSlide*2.

Base case: If *via* is Empty then in fact *varSlide*1 = *varSlide*2, therefore *slide*1 = *slide*2 which means that *slide*2 is reachable from *slide*1.

via is Extend(prefix′, n′), then n' is reachable from *varSlide*1 using the path $prefix'$, and n' is a predecessor of *varSlide*2. Let n'' be the last regular varSlide on the VarSlideDGPath between *varSlide*1 and *varSlide*2 (could be n' or another varSlide if n' is Phi), and let $prefix''$ be the VarSlideDGPath that is $prefix'$ without n' (meaning up until n''). Let n be the corresponding slide of n''. We need to show that n is reachable from *slide*1 and that *slide*2 is reachable from n.

Inductive hypothesis: Suppose n is reachable from *slide*1.

Inductive step: *varSlide*2 is reachable from n'', therefore *slide*2 is reachable from n using Lemma 6 (as explained next). We have shown a path between *slide*1 and n and between n and *slide*2, therefore there is a path between *slide*1 and *slide*2 and we can say that *slide*2 is reachable from *slide*1. □

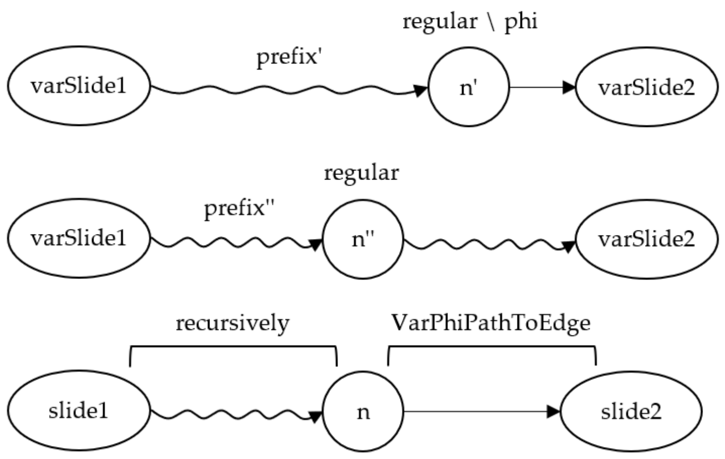

FIGURE 3.3: PathBackCorrespondence demonstration.

Lemma 6 (Var phi path to edge).

```
lemma VarPhiPathToEdge(slide1: Slide, slide2: Slide,
slideDG: SlideDG, varSlideDG: VarSlideDG, S: Statement,
S': Statement, XLs: seq<set<Variable>>, X: seq<Variable>)
  requires Valid(S) ∧ Core(S)
  requires Valid(S') ∧ Core(S')
  requires IsSlideDGOf(slideDG, S)
  requires IsVarSlideDGOf(varSlideDG, S)
  requires VarSlideTag(VarSlideOf(S, S', slide1, XLs, x)) = Regular
  requires VarSlideTag(VarSlideOf(S, S', slide2, XLs, x)) = Regular
  requires VarSlideDGReachablePhi(varSlideDG,
    VarSlideOf(S, S', slide1, XLs, x),
    VarSlideOf(S, S', slide2, XLs, x),
    VarSlideDGVarSlides(varSlideDG))
  requires ValidXLs(glob(S), XLs, X)
  requires RemoveEmptyAssignments(Rename(S', XLs, X, glob(S))) = S
  ensures SlideDependence(slide1, slide2, S)
```

Proof. Let *vSlide1*, *varL1* and *vSlide2*, *varL2* be the corresponding var-slides and labels of *slide1*, *l1* and *slide2*, *l2*, respectively. Let v be the variable of the instance defined in *vSlide1* and used in *vSlide2*. Let v'' be the instance of v defined in *vSlide1* and v' be the instance of v used in *vSlide2*, which is live on entry in *varL2*.

If $v'' = v'$, then there is an edge between *vSlide1* and *vSlide2* in varSlideDG and an edge between *slide1* and *slide2* in slideDG, meaning *slide2* is slide-dependent on *slide1*.

If v'' != v', then v' is Phi (recall that v' is live on entry in *varL2*). Using Theorem 2, we can say that all varSlides of RD_IN_FOR(S, $l2$, v) = RF(v'). v'' is a regular predecessor of v', meaning it is in RF(v'), therefore its slide (v, $l1$) is in RD_IN_FOR(S, $l2$, v). Thus, *slide2* is slide-dependent on *slide1*. □

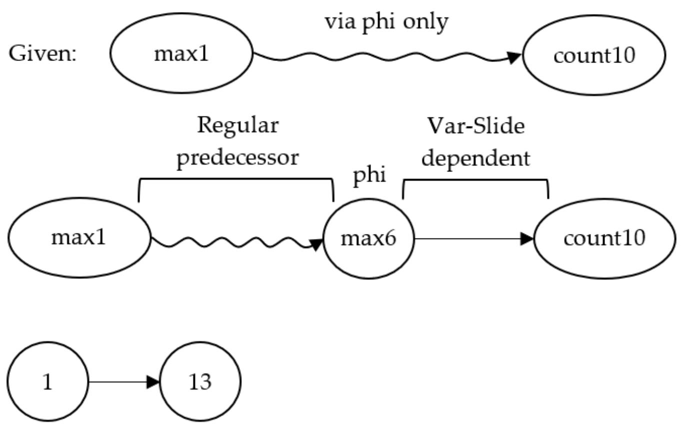

FIGURE 3.4: VarPhiPathToEdge example: max1 as v'', max6 as v', slide 1 as *slide1* and slide 13 as *slide2*.

Chapter 4

A novel slide-based slicing algorithm

In this section we describe our slide-based slicing algorithm and provide examples.

4.1 The algorithm

Given a statement S, a set of variables V, and a slide-dependence graph *slideDG* of S, our algorithm computes the set of slides whose union is the program's slice of S for the values of each variable in V when exiting S.

Let S_V be an initial set of final-def slides of S on V, and let *WorkSet* be a temporary initialized to S_V. *Poll* returns a slide and removes it from *WorkSet*, and for each such slide in *WorkSet* we find its slide-dependence predecessors in the slideDG (that are not in S_V) and add them to S_V and to *WorkSet*. The algorithm terminates when there are no slides left in *WorkSet*, and it returns S_V.

The algorithm has a linear time-complexity in the size of the slide-dependence graph of S. Let N be the number of nodes and E be the number of edges in the graph, in the worst-case *WorkSet* contains all slides (N) and the algorithm finds all predecessors of all slides (E), therefore the algorithm's worst-case time complexity is O(N+E).

The algorithm is presented as Algorithm 1:

```
Result: $S_V$
$S_V := \bigcup_{v \in V}\{$final-def slides of $S$ on $v$ in $slideDG\}$;
$WorkSet := S_V$;
while $WorkSet \neq \emptyset$ do
    $S_n := Poll(WorkSet)$;
    $NewlyReachable := \{Predecessors\ of\ S_n\ in\ slideDG\} - S_V - \{S_n\}$;
    $S_V := S_V \cup NewlyReachable$;
    $WorkSet := WorkSet \cup NewlyReachable$;
end
```

Algorithm 1: Slide-based slicing algorithm

4.2 Examples

In the first example our program computes the sum of two variables (see Listing 4.1), and $V = \{w\}$ where V contains a variable that is not included in the code. S_V is empty (since there are no final-def slides of S on w), therefore *WorkSet* is empty and the loop will not execute (see Listing 4.2).

```
1  x := 5;
2  y := 4;
3  z := x + y
```

LISTING 4.1: Program S1

```
1  skip
```

LISTING 4.2: *ComputeSlice*(S1,{w})

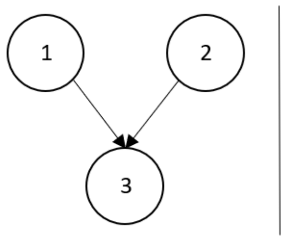

FIGURE 4.1: The slideDG of S1 (from Listing 4.1) on the left, and the slideDG of its slice (from Listing 4.2) on the right.

In the second example our program calculates the sum and product for an array of numbers (see Listing 4.3) and our set of variables is V = {sum}. S_V and *WorkSet* will initially contain the final-def slides of *sum*: slides 2 and 5. Then, we only add the predecessors of slide 5 to S_V (there are no predecessors for slide 2): slides 1 and 7 (see the slideDG in Figure 4.2 for the predecessors). Again, there are no predecessors for slides 1 and 7, therefore the slice of *sum* is {1,2,5,7} (see Listing 4.4).

```
1  i := 0;
2  sum := 0;
3  prod := 1;
4  while i < a.length do
5    sum := sum + a[i];
6    prod := prod * a[i];
7    i := i + 1
8  od
```

LISTING 4.3: Program S2

```
1  i := 0;
2  sum := 0;
3
4  while i < a.length do
5    sum := sum + a[i];
6
7    i := i + 1
8  od
```

LISTING 4.4: *ComputeSlice*(S2,{sum})

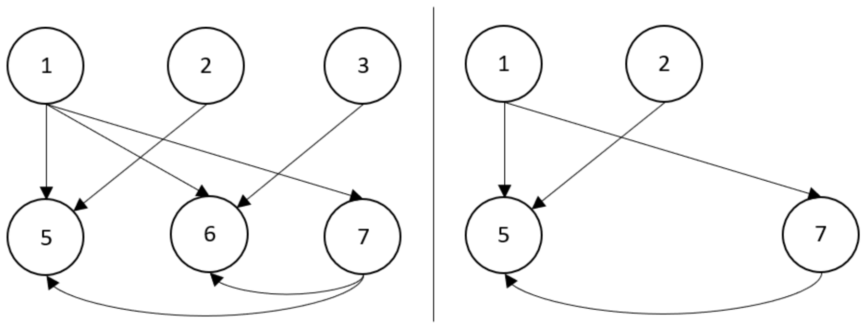

FIGURE 4.2: The slideDG of S2 (from Listing 4.3) on the left, and the slideDG of its slice (from Listing 4.4) on the right.

In the third and final example, our program sums the values of an array in even indices or in odd incides, depending on the parity of the array's length, respectively (see Listing 4.5). Let $V = \{sum\}$, therefore all of the programs slides {1,3,5,8,9} will be in the resulting slice (see Listing 4.6).

```
1   sum := 0;
2   if a.length % 2 == 0 then
3     i := 0
4   else
5     i := 1
6   fi;
7   while i < a.length do
8     sum := sum + a[i];
9     i := i + 2
10  od
```

LISTING 4.5: Program S3

```
1   sum := 0;
2   if a.length % 2 == 0 then
3     i := 0
4   else
5     i := 1
6   fi;
7   while i < a.length do
8     sum := sum + a[i];
9     i := i + 2
10  od
```

LISTING 4.6: *ComputeSlice*(S3,{sum})

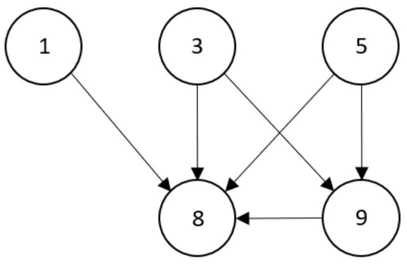

FIGURE 4.3: The slideDG of S3 (from Listing 4.5) and the slideDG of its slice on {sum} (from Listing 4.6) are the same.

Chapter 5

Proof of correctness

In this section we provide the correctness proof of our slide-based slicing algorithm.

5.1 Our algorithm

The SSA-based algorithm is semantics-preserving, as proven in [7]. Our goal is to prove that the resulting statements of the two algorithms are textually identical, which means that our algorithm also results in a semantics-preserving slice. To prove this we use the following theorem:

Theorem 7 (Identical slices)**.** Given three statements S, S1, and S2, where S1 is the slide-based slice of S, S2 is the SSA-based slice of S, and there are no self or empty assignments in S, then S1 is textually identical to S2.

Proof. Instead of using S1 and S2 we can use slipOf(S1, []) and slipOf(S2, []), therefore we apply Lemma 10 which ensures that slipOf(S1, []) = slipOf(S2, []). The proof of Lemma 10 is provided later in this chapter. □

The following terms, definitions (10, 11), and lemmas (8, 9) are needed for the proof of Lemma 10.
We begin with presenting the relevant terminology:

- S: The original statement.
- V: The set of variables for the slice.
- S': The SSA version of S.
- V': The set of the live-on-exit instance of each variable in V.
- SV': The result of ComputeFISlice(S',V').
- *res*: The SSA-based slice of S on V.
- SV: The slide-based slice of S on V.
- *slidesSV*: The set of slides of SV.
- *varSlidesSV*: The set of varSlides of SV'.

We now continue with two formal definitions:

Definition 10 (slidesSV)**.** Given a statement S, a set of variables V, and a slide-dependence graph slideDG:

```
∀ Sm • Sm in SlideDGSlides(slideDG)
∧ (∃ Sn • Sn in finalDefSlides(S, slideDG, V)
∧ SlideDGReachable(slideDG, Sm, Sn, SlideDGSlides(slideDG))))
```

The set of slides of S in which each slide has a path to a final-def slide in the slideDG of S. In our example, V = {isSorted} then slidesSV consists of slides 1, 3, 4, 7, 13, 17, 20 and 22 (where 20 and 22 are the final-def slides).

Definition 11 (varSlidesSV)**.** Given a statement S, a set of instances V′, and a varSlide-dependence graph varSlideDG:

```
∀ Sm′ • Sm′ in VarSlideDGVarSlides(varSlideDG) ∧
(∃ Sn′ • VarSlideVariable(Sn′) in V′ ∧
VarSlideDGReachable(varSlideDG, Sm′, Sn′,
VarSlideDGVarSlides(varSlideDG)))
```

The set of varSlides of S′ in which each varSlide has a path to a live-on-exit instance (in V′) in the varSlideDG of S′. In our example, V′ = {isSorted12} then varSlidesSV consists the varSlides of max1, i3, count4, max5, i5, count5, max6, max7, count9, count10, i11, isSorted12, isSorted13 and isSorted14.

We show the correspondence between slidesSV and varSlidesSV with the following lemma:

Lemma 8 (LemmaSlidesSVToVarSlidesSV)**.** Given a statement S, its corresponding SSA version S′, and the mapping between X and XLs, for each slide in the slideDG of S we have:

```
slide in slidesSV ⟺
  VarSlideOf(S, S′, slide, XLs, X) in varSlidesSV
```

Proof. We prove the first direction of the lemma:

```
slide in slidesSV ⟹
  VarSlideOf(S, S′, slide, XLs, X) in varSlidesSV
```

For each *slide* ∈ *slidesSV*, we find its corresponding varSlide in S′, and show that it is also in *varSlidesSV*. Let *vSlide* be the corresponding varSlide of *slide* using the function call *vSlide := VarSlideOf(S, S′, slide, XLs, X)*. This ensures that for each slide there is exactly one corresponding varSlide (and backwards). In order to prove that *vSlide* ∈ *varSlidesSV* we need to show that there exists an instance in V′ whose varSlide is reachable from *vSlide* (Definition 11). Let:

- *finalDefSlide*: A final-def slide that is reachable from *slide* (could be that *slide* is in fact *finalDefSlide*). The final-def slides in our example (where V={isSorted}) are slides 20 and 22.
- *finalDefVarSlide*: The varSlide Of *finalDefSlide*. The final-def varSlides in our example are the varSlides of isSorted13 and isSorted14.
- *lvExitVarSlide*: The varSlide of v' (where v' is the instance of the variable of *finalDefSlide* that is in V'), which we need to prove is reachable from *vSlide*. Note that v' is a live on exit instance of S'. In our example it is the varSlide of isSorted12.

Using Theorem 1 where *varLabel* is [] (because v' is live on exit from S'), we can say that all varSlides of RD_OUT_FOR(S, [], v) are RF(v'). Using Definition 7, these varSlides are the corresponding varSlides of the final-def slides.

If *lvExitVarSlide* is Regular, then RF(v') is {*lvExitVarSlide*} and {*finalDefVarSlide*} is in {*lvExitVarSlide*}.

If *lvExitVarSlide* is Phi, then RF(v') is the set of regular predecessors of v', which includes *finalDefVarSlide*. Thus, we can conclude that there is a path from *finalDefVarSlide* to *lvExitVarSlide*, therefore *lvExitVarSlide* is reachable from *finalDefVarSlide*.

In our example, the instance of *lvExitVarSlide* is *isSorted12* and the set reaching out from the corresponding point is *S* is {(isSorted,20), (isSorted,22)}. Using Theorem 1, the varSlides of this set are {isSorted13, isSorted14} and are the regular-predecessor varSlides of the varSlide of isSorted12.
Now we prove the second direction of the lemma:

```
VarSlideOf(S, S', slide, XLs, X) in varSlidesSV ⟹
  slide in slidesSV
```

or in other words:

```
slide ∉ slidesSV ⟹
  VarSlideOf(S, S', slide, XLs, X) ∉ varSlidesSV
```

Suppose *slide* $\notin$ *slidesSV*, let us assume for the sake of contradiction that *vSlide* $\in$ *varSlidesSV*. By Definition 10, there exists a varSlide, *lvExitVarSlide*, whose variable is in V' and is reachable from *vSlide*. Recall that *vSlide* is a regular varSlide (formed from the *VarSlideOf* function). Let *Sn'* be the last regular varSlide on a VarSlideDG-Path between *vSlide* and *lvExitVarSlide* (including *lvExitVarSlide*), and let *Sn* be its corresponding slide. Again, the instance of *lvExitVarSlide* is live-on-exit from S', and by using Definition 7 the set RD_OUT(S, []) from the corresponding label [] in S are final-def slides of S. By Theorem 1, the varSlides of this set are regular-predecessors of *lvExitVarSlide*, therefore *Sn* is a final-def slide of S. By using Theorem 4 (PathBack-Correspondence, as explained in Chapter 3) *Sn* is reachable from *slide*. According to Definition 10, *slide* $\in$ *slidesSV*, in contradiction to *slide* $\notin$ *slidesSV*, hence *vSlide* $\notin$ *varSlidesSV*.

□

We have proved that each *slide* in *slidesSV* has a corresponding *vSlide* in *varSlidesSV*. We also gained another property: |*slidesSV*| = |*varSlidesSV*|, by counting exactly one varSlide for each slide using the *VarSlideOf* function.

Next, we define the following predicate:

```
predicate MatchingSlips(S1: Statement, S2: Statement, l: Label)
  reads *
  requires Valid(S1) ∧ Valid(S2)
  requires Core(S1) ∧ Core(S2)
  requires ValidLabel(l, S1)
  requires ValidLabel(l, S2)
{
  var slipOfS1 := slipOf(S1, l);
  var slipOfS2 := slipOf(S2, l);

  match slipOfS1 {
  case Skip ⇒
    IsSkip(slipOfS2)
  case Assignment(LHS,RHS) ⇒
    IsSkip(slipOfS2) ∨ IsAssignment(slipOfS2)
  case SeqComp(S1,S2) ⇒
    IsSkip(slipOfS2) ∨ IsSeqComp(slipOfS2)
  case IF(B,Sthen,Selse) ⇒
```

```
    IsSkip(slipOfS2) ∨ IsIF(slipOfS2)
  case DO(B,Sloop) ⇒
    IsSkip(slipOfS2) ∨ IsDO(slipOfS2)
  }
}
```

We use this predicate for *S* and *res* in order to prove the following lemma:

Lemma 9 (LemmaMatchingSlips)**.** Given a statement S, its SSA-based slice res, and a valid label in both statements l:

```
MatchingSlips(S, res, l)
```

Proof. Let *slipOfS* be the result of slipOf(S, l), and *slipOfRes* the result of slipOf(res, l). For this proof we distinguish between the various values of slipOfS.

- If slipOfS is Skip then by converting Skip to SSA, computing its FI-Slice and converting back to SSA - it will stay Skip, therefore slipOfRes is also Skip.
- If slipOfS is Assignment, by converting it to SSA it will stay an Assignment. After computing its FI-Slice it will stay Assignment or become Skip (if it is not in the slice). Finally, the result of converting back from SSA will Assignment or Skip.
- If slipOfS is Sequential Composition, by converting it to SSA it will stay Sequential Composition. After computing its FI-Slice it will stay Sequential Composition or become Skip (if it is not in the slice). Finally, the result of converting back from SSA will be either Sequential Composition or Skip.
- If slipOfS is IF, by converting it to SSA it will stay as IF. After computing its FI-Slice it will stay IF or become Skip (if it is not in the slice). Finally, the result of converting back from SSA will be either IF or Skip.
- If slipOfS is DO, by converting it to SSA it will be Sequential Composition of Phi-Assignment and DO. After computing its FI-Slice the Sequential Composition will stay Sequential Composition or become Skip (if it is not in the slice). Finally, the result of converting back from SSA will return to be either DO or Skip.

□

Finally, in order to show that slipOf(SV, []) = slipOf(res, []) (and complete the proof of Theorem 7) we need to prove the following lemma:

Lemma 10 (LemmaIdenticalSlips)**.** Given the slide-based algorithm slice SV, the SSA-based algorithm slice res (both for a statement S and a set of variables V), and a valid label in both slices l:

```
slipOf(SV, l) = slipOf(res, l)
```

Proof. We start by using Lemma 11 (full definition and proof in Appendix A.2.3), which states that: IsSkip(slipOf(SV, l)) $\iff$ IsSkip(slipOf(res, l)). If slipOf(SV, l) is Skip, then slipOf(res, l) is Skip; if slipOf(SV, l) is not Skip, then slipOf(S, l) is also not Skip (SV is a substatement of S) and we use Lemma 9 to determine that slipOf(SV, l) and slipOf(res, l) (and in fact slipOf(S, l)) are of the same type. We prove this lemma by induction on l:
Base case:

- slipOf(SV, l) is Skip: slipOf(res, l) is also Skip, as explained using Lemma 11.
- slipOf(SV, l) is Assignment: slipOf(res, l) is also Assignment (as explained above) and we need to show that it is exactly the same assignment. First, we find the set of slides of all the assignments in slipOf(SV, l) and the set of varSlides of all the assignments in slipOf(SV′, l′) (where l' is the corresponding label of l in SV') and mark them as A and B, respectively. We already know that $|slidesSV| = |varSlidesSV|$, and that each *slide* in *slidesSV* has a corresponding *varSlide* in *varSlidesSV* (and backwards). We also know that A is a subset of *slidesSV* and B is a subset of *varSlides*, therefore $|A| = |B|$. Then, for each slide and variable in the first set, there is a corresponding varSlide and instance in the second set (Lemma 8), and if we convert each of those instances back to their original variables we will get the assignment of res (which is the exact same assignment of SV).

Inductive hypothesis:

- If slipOf(SV, l) is a Sequential Composition statement, suppose slipOf(SV, l+[1]) = slipOf(res, l+[1]) and slipOf(SV, l+[2]) = slipOf(res, l+[2]) (both branches of the Sequential Composition statement).
- If slipOf(SV, l) is an If statement, suppose slipOf(SV, l+[1]) = slipOf(res, l+[1]) and slipOf(SV, l+[2]) = slipOf(res, l+[2]) (both branches of the If statement).
- If slipOf(SV, l) is a Do statement, suppose slipOf(SV, l+[1]) = slipOf(res, l+[1]) (loop body of the Do statement).

Inductive step:

- If slipOf(SV, l) is a Sequential Composition statement, slipOf(res, l) is of the same type (as explained above). By our hypothesis, slipOf(SV, l) = slipOf(res, l).
- If slipOf(SV, l) is an If statement, slipOf(res, l) is of the same type (as explained above). Let b be the boolean expression in slipOf(S, l). In the transition to SSA, the variables in b were renamed into new instances in slipOf(S′, l′), where l′ is the result of VarLabelOf function on l. Then, after computing its flow-insensitive slice, b remained the same exact expression in slipOf(SV′, l′). In the transition back from SSA, the instances in b were renamed to their original variables in slipOf(res, l), where l is the result of LabelOf function on l′ (and using Lemma 12, in Appendix A.2.3), it is the original l. Therefore b remained the same exact boolean expression in slipOf(res, l) as in slipOf(S, l). By our hypothesis, slipOf(SV, l) = slipOf(res, l).
- If slipOf(SV, l) is a Do statement, slipOf(res, l) is of the same type (as explained above). The boolean expression in slipOf(res, l) is the same exact boolean expression in slipOf(S, l), as previously explained. By our hypothesis, slipOf(SV, l) = slipOf(res, l).

□

To summarize, we proved that our slide-based algorithm is semantics-preserving. This was done by showing that the SSA-based slice of S is textually identical to the slide-based slice of S.

Chapter 6

Conclusion

In this thesis we have presented a new slide-based slicing algorithm. We proved that the resulting slice of the algorithm is textually identical to the resulting slice of an existing SSA-based slicing algorithm [7], which is semantics-preserving, therefore our algorithm is also semantics-preserving. Our algorithm is syntax-preserving, therefore we gained syntax-preservation for the SSA-based algorithm. We have also discussed how we improved the complexity of the SSA-based algorithm using our algorithm.

We have formalized the slide-dependence graph used in our slicing algorithm, and formalized the varSlide-dependence graph and the connection between the two graphs in order to prove that both resulting slices are textually identical.

We have also formalized the transition of a program into SSA, the computation of its flow-insensitive slice, and the transition back from SSA. Then we showed that the resulting slice of every slicing algorithm that meets these requirements is syntax-preserving and textually identical to our algorithms resulting slice.

In this thesis we have made some simplifying assumptions. Our algorithm computes a slice for a statement of a rather simple language and does not support complex statements such as goto or switch statements, as we wanted to prove the textual equivalence to the code in [7]. Moreover, we can use slideDG or PDG as a program representation also for complex statements (as been done in [4] and [8]) and forgoing the semantic-preservation of the statements. Also, some of the lemmas needed in the proof of our algorithm have been used with the Dafny language without providing a formal proof, due to the scope of this thesis.

Some directions for further research are:

- Implementing a co-slicing algorithm (described in [7]), an advanced sliding transformation in which the complement reuses a selection of extracted results, thus yielding a potentially smaller complement.
- Using our formal framework in order to prove the correctness of a PDG-based slicing algorithm.
- Expanding our algorithm for slicing from a different point in the statement. For a statement S and a set of variables V, our algorithm uses backward slicing from the final-def nodes of each variable in V. This could be expanded by slicing from any regular node and each variable, not necessarily its final-def.
- Using our formal framework to develop algorithms for cloning, code-motion refactoring, etc.

We begun our work by developing a slicing algorithm that is using slide-dependence graph for program representation. In order to prove our algorithm we have developed a solid formal framework that, as described above, can be used for many other applications that improve code quality.

Appendix A

Appendix

A.1 Full definitions

A.1.1 Utility functions

```
predicate Valid(S: Statement)
{
  match S {
  case Skip ⇒ true
  case Assignment(LHS,RHS) ⇒ ValidAssignment(LHS,RHS)
  case SeqComp(S1,S2) ⇒ Valid(S1) ∧ Valid(S2)
  case IF(B0,Sthen,Selse) ⇒
  (∀ state: State • B0.0.requires(state)) ∧
  Valid(Sthen) ∧ Valid(Selse)
  case DO(B,Sloop) ⇒
  (∀ state: State • B.0.requires(state)) ∧ Valid(Sloop)
  } ∧
  ∀ state1: State, P: Predicate • P.0.requires(state1)
}

predicate Core(stmt: Statement)
{
  match stmt {
  case Skip ⇒ true
  case Assignment(LHS, RHS) ⇒ true
  case SeqComp(S1, S2) ⇒ Core(S1) ∧ Core(S2)
  case IF(B0,Sthen,Selse) ⇒ Core(Sthen) ∧ Core(Selse)
  case DO(B,Sloop) ⇒ Core(Sloop)
  }
}

function method setOf<T>(s: seq<T>): (res: set<T>)
  ensures ∀ v • v in res ⟺ v in s
{
  set x | x in s
}

predicate ValidAssignment(LHS: seq<Variable>, RHS: seq<Expression>)
{
  |LHS| = |RHS| ∧ |setOf(LHS)| = |LHS|
}

predicate ValidLabel(l: Label, S: Statement)
  reads *
  requires Valid(S) ∧ Core(S)
{
  if l = [] then true
  else
  match S {
```

```
  case Skip ⇒ false
  case Assignment(LHS,RHS) ⇒ false
  case SeqComp(S1,S2) ⇒
    if l[0] = 1 then ValidLabel(l[1..], S1)
    else ValidLabel(l[1..], S2)
  case IF(B0,Sthen,Selse) ⇒
    if l[0] = 1 then ValidLabel(l[1..], Sthen)
    else ValidLabel(l[1..], Selse)
  case DO(B,Sloop) ⇒
    if l[0] = 1 then ValidLabel(l[1..], Sloop)
    else false
  }
}

function method def(S: Statement): set<Variable>
{
  match S {
  case Assignment(LHS,RHS) ⇒ setOf(LHS)
  case Skip ⇒ {}
  case SeqComp(S1,S2) ⇒ def(S1) + def(S2)
  case IF(B0,Sthen,Selse) ⇒ def(Sthen) + def(Selse)
  case DO(B,Sloop) ⇒ def(Sloop)
  }
}

function method glob(S: Statement): set<Variable>
{
  set x | x in def(S) + input(S)
}

function GetRHSVariables(seqExp: seq<Expression>): set<Variable>
{
  if seqExp = [] then {}
  else seqExp[0].1 + GetRHSVariables(seqExp[1..])
}
```

A.1.2 Slides functions

```
function SlideLabel(s: Slide): Label { s.0 }

function SlideVariable(s: Slide): Variable { s.1 }

function SlideLabels(s: Slide, S: Statement): set<Label>
  requires Valid(S) ∧ Core(S)
  requires s in SlidesOf(S, def(S))
{
  set l | l = SlideLabel(s) ∨
  (l < SlideLabel(s) ∧ (IsDO(slipOf(S, l)) ∨ IsIF(slipOf(S, l))))
}

function SlidesOf(S: Statement, V: set<Variable>)
  : set<Slide>
  reads *
  requires Valid(S) ∧ Core(S)
  ensures ∀ s • s in SlidesOf(S, V) ⟹
    ValidLabel(SlideLabel(s), S) ∧
    ¬IsEmptyAssignment(slipOf(S, SlideLabel(s)))
{
  SlidesOfRec(S, V, [])
}

function SlidesOfRec(S: Statement, V: set<Variable>,
  l: Label): (slides: set<Slide>)
  reads *
  requires Valid(S) ∧ requires Core(S)
  ensures ∀ s • s in slides ⟹
    ValidLabel(SlideLabel(s), S) ∧
    ¬IsEmptyAssignment(slipOf(S, SlideLabel(s)))
{
  match S {
  case Skip ⇒ {}
  case Assignment(LHS,RHS) ⇒
    set v | v in V * setOf(LHS) • (l, v)
  case SeqComp(S1,S2) ⇒
    SlidesOfRec(S1, V, l+[1]) + SlidesOfRec(S2, V, l+[2])
  case IF(B0,Sthen,Selse) ⇒
    SlidesOfRec(Sthen, V, l+[1]) + SlidesOfRec(Selse, V, l+[2])
  case DO(B,Sloop) ⇒
    SlidesOfRec(Sloop, V, l+[1])
  }
}

function UsedVars(S: Statement, l: Label): set<Variable>
  requires Valid(S) ∧ Core(S)
  requires ValidLabel(l, S)
{
  var slipOfS := slipOf(S, l);

  match S {
  case Assignment(LHS,RHS) ⇒ set v | v in GetRHSVariables(RHS)
  case SeqComp(S1,S2) ⇒ {}
  case IF(B,Sthen,Selse) ⇒ set v | v in B.1
  case DO(B,S0) ⇒ set v | v in B.1
  case Skip ⇒ {}
  }
}

function GetRHSVariables(seqExp: seq<Expression>): set<Variable>
{
  if seqExp = [] then {}
  else seqExp[0].1 + GetRHSVariables(seqExp[1..])
}

function SlideDGStatement(slideDG: SlideDG): Statement { slideDG.0 }

function SlideDGSlides(slideDG: SlideDG): set<Slide> { slideDG.1 }

function SlideDGMap(slideDG: SlideDG):
  map<Slide, set<Slide>> { slideDG.2 }
```

A.1.3 VarSlides functions

```
function VarSlideVariable(s: VarSlide): Variable { s.0 }

function VarSlideTag(s: VarSlide): VarSlideTag { s.1 }

function VarSlideLabels(s: VarSlide, S: Statement): set<Label>
  requires Valid(S) ∧ Core(S)
  requires s in VarSlidesOf(S, def(S))
{
  var assignmentLabels := VarSlideAssignmentLabels(s, S, []);

  assignmentLabels +
  set l | ((∀ l' • l' in assignmentLabels ⟹ l < l') ∧
  (IsDO(slipOf(S, l)) ∨ IsIF(slipOf(S, l))))
}

function VarSlideAssignmentLabels(s: VarSlide, S: Statement,
  l: Label): set<Label>
  reads *
  requires Valid(S) ∧ Core(S)
  requires ValidLabel(l, S)
  requires s in VarSlidesOf(S, def(S))
{
  match S {
  case Assignment(LHS,RHS) ⇒
    if VarSlideVariable(s) in LHS then {l} else {}
  case Skip ⇒ {}
  case SeqComp(S1,S2) ⇒
    VarSlideAssignmentLabels(s, S1, l + [1]) +
    VarSlideAssignmentLabels(s, S2, l + [2])
  case IF(B0,Sthen,Selse) ⇒
    VarSlideAssignmentLabels(s, Sthen, l + [1]) +
    VarSlideAssignmentLabels(s, Selse, l + [2])
  case DO(B,Sloop) ⇒
    VarSlideAssignmentLabels(s, Sloop, l + [1])
  }
}

function VarSlidesOf(S': Statement, V: set<Variable>):
  set<VarSlide>
  reads *
  requires Valid(S') ∧ Core(S')
{
  match S'
  case Skip ⇒ {}
  case Assignment(LHS,RHS) ⇒
    set v | v in setOf(LHS) • (v, Regular)
  case SeqComp(S1,S2) ⇒
    if IsDO(S2) then
      assert IsAssignment(S1);
      match S1
      case Assignment(LHS,RHS) ⇒
        (set v | v in setOf(LHS) • (v, Phi)) + VarSlidesOf(S2, V)
    else
      VarSlidesOf(S1, V) + VarSlidesOf(S2, V)
  case IF(B0,Sthen,Selse) ⇒
    assert IsSeqComp(Sthen);
    match Sthen
    case SeqComp(S1,S2) ⇒
      assert IsAssignment(S2);
      match S2
      case Assignment(LHS,RHS) ⇒
        ((set v | v in setOf(LHS) • (v, Phi)) + VarSlidesOf(S1, V) +
        assert IsSeqComp(Selse);
        match Selse
        case SeqComp(S1',S2') ⇒
          assert IsAssignment(S2');
          match S2'
          case Assignment(LHS',RHS') ⇒
            (set v | v in setOf(LHS') • (v, Phi)) + VarSlidesOf(S1', V))
  case DO(B,Sloop) ⇒
    assert IsSeqComp(Sloop);
    match Sloop
    case SeqComp(S1,S2) ⇒
      assert IsAssignment(S2);
      match S2
      case Assignment(LHS,RHS) ⇒
        (set v | v in setOf(LHS) • (v, Phi)) + VarSlidesOf(S1, V)
}

function VarSlideDGStatement(varSlideDG: VarSlideDG):
  Statement { varSlideDG.0 }

function VarSlideDGVarSlides(varSlideDG: VarSlideDG):
  set<VarSlide> { varSlideDG.1 }

function VarSlideDGMap(varSlideDG: VarSlideDG):
  map<VarSlide, set<VarSlide>> { varSlideDG.2 }
```

A.1.4 Graphs correspondence functions

```
function VarLabelOf(S: Statement, S': Statement, l: Label,
  XLs: seq<set<Variable>>, X: seq<Variable>): Label
  reads *
  requires Valid(S) ∧ Valid(S')
  requires Core(S) ∧ Core(S')
  requires ValidLabel(l, S)
  requires ValidXLs(glob(S), XLs, X)
  requires S = RemoveEmptyAssignments(Rename(S', XLs, X, glob(S)))
  requires MatchingSlipsToSSA(S, [], S', [])
  ensures ValidLabel(VarLabelOf(S, S', l, XLs, X), S')
{
  match S {
  case Skip ⇒
    assert IsSkip(S');
    assert l = [];
    []
  case Assignment(LHS,RHS) ⇒
    assert IsAssignment(S');
    assert l = [];
    []
  case SeqComp(S1,S2) ⇒
    assert IsSeqComp(S');
    if l = [] then [] else
      match S' {
      case SeqComp(S1',S2') ⇒
        if l[0] = 1 then [1] + VarLabelOf(S1, S1', l[1..], XLs, X)
        else [2] + VarLabelOf(S2, S2', l[1..], XLs, X)
      }
  case IF(B0,Sthen,Selse) ⇒
    assert IsIF(S');
    if l = [] then [] else
      match S' {
      case IF(B0',Sthen',Selse') ⇒
        if l[0] = 1 then [1,1] + VarLabelOf(Sthen, Sthen', l[1..], XLs, X)
        else [2,1] + VarLabelOf(Selse, Selse', l[1..], XLs, X)
      }
  case DO(B,Sloop) ⇒
    assert IsSeqComp(S');
    if l = [] then [] else
      match S' {
      case SeqComp(S1',S2') ⇒
        assert IsDO(S2');
        match S2' {
        case DO(B',Sloop') ⇒
          [2,1,1] + VarLabelOf(Sloop, Sloop', l[1..], XLs, X)
        }
      }
  }
}

predicate MatchingSlipsToSSA(S: Statement, l: Label,
  S': Statement, l': Label)
  reads *
  requires Valid(S) ∧ Valid(S')
  requires Core(S) ∧ Core(S')
  requires ValidLabel(l, S)
  requires ValidLabel(l', S')
{
  var slipOfS := slipOf(S, l);
  var slipOfS' := slipOf(S', l');

  match slipOfS {
  case Skip ⇒ IsSkip(slipOfS')
  case Assignment(LHS,RHS) ⇒ IsAssignment(slipOfS')
  case SeqComp(S1,S2) ⇒ IsSeqComp(slipOfS')
  case IF(B,Sthen,Selse) ⇒ IsIF(slipOfS') ∧
    match slipOfS' {
    case IF(B',Sthen',Selse') ⇒
      IsSeqComp(Sthen') ∧ IsSeqComp(Selse')
    }
  case DO(B,Sloop) ⇒ IsSeqComp(slipOfS') ∧
    match slipOfS' {
    case SeqComp(S1',S2') ⇒
      IsAssignment(S1') ∧ IsDO(S2') ∧
      match S2' {
      case DO(B', Sloop') ⇒ IsSeqComp(Sloop')
      }
    }
  }
}

predicate MatchingSlipsFromSSA(S': Statement, l': Label,
  S: Statement, l: Label)
  reads *
  requires Valid(S) ∧ Valid(S')
  requires Core(S) ∧ Core(S')
  requires ValidLabel(l, S)
  requires ValidLabel(l', S')
{
  MatchingSlipsToSSA(S, l, S', l')
}

function InstanceOf(S': Statement, l': Label, v: Variable,
  XLs: seq<set<Variable>>, X: seq<Variable>,
  globS: set<Variable>): Variable
  reads *
  requires Valid(S') ∧ Core(S')
  requires ValidXLs(globS, XLs, X)
  requires ValidLabel(l', S')
{
  if l' = [] then
    assert IsAssignment(S');
    var v' :| v' in setOf(GetLHS(S')) * InstancesOf(S', v, X, XLs, globS);
    v'
  else
    match S' {
    case SeqComp(S1',S2') ⇒
      if l'[0] = 1 then InstanceOf(S1', l'[1..], v, XLs, X, globS)
      else InstanceOf(S2', l'[1..], v, XLs, X, globS)
    case IF(B0',Sthen',Selse') ⇒
```

```
      if l'[0] = 1 then InstanceOf(Sthen', l'[1..], v, XLs, X, globS)
      else InstanceOf(Selse', l'[1..], v, XLs, X, globS)
    case DO(B',Sloop') ⇒
      InstanceOf(Sloop', l'[1..], v, XLs, X, globS)
    }
}

function InstancesOf(S: Statement, v: Variable, X: seq<Variable>,
  XLs: seq<set<Variable>>, globS: set<Variable>): set<Variable>
  requires ValidXLs(globS, XLs, X)
{
  if X = [] then {}
  else if X[0] = v then XLs[0]
  else InstancesOf(S, v, X[1..], XLs[1..], globS)
}

predicate ValidXLs(globS: set<Variable>, XLs: seq<set<Variable>>,
  X: seq<Variable>)
{
  |X| = |XLs| ∧
  (∀ i,j • 0 ≤ i < j < |XLs| ⟹ XLs[i] ∩̸ XLs[j]) ∧
  (∀ s • s in XLs ⟹ s ∩̸ globS)
}

function Rename(S': Statement, XLs: seq<set<Variable>>, X: seq<Variable>,
  globS: set<Variable>): Statement
  reads *
  requires Valid(S') ∧ Core(S')
  requires ValidXLs(globS, XLs, X)
  ensures Valid(Rename(S', XLs, X, globS)) ∧
    Core(Rename(S', XLs, X, globS))
{
  match S' {
  case Assignment(LHS,RHS) ⇒
    RenameAssignment(LHS, RHS, XLs, X, globS)
  case SeqComp(S1,S2) ⇒
    SeqComp(Rename(S1, XLs, X, globS), Rename(S2, XLs, X, globS))
  case IF(B0,Sthen,Selse) ⇒
    IF(RenameBoolExp(B0, XLs, X),
    Rename(Sthen, XLs, X, globS), Rename(Selse, XLs, X, globS))
  case DO(B,S1) ⇒
   DO(RenameBoolExp(B, XLs, X), Rename(S1, XLs, X, globS))
  case Skip ⇒ Skip
  }
}

function RemoveEmptyAssignments(S: Statement): Statement
  requires Core(S) ∧ Valid(S)
{
  match S {
  case Assignment(LHS,RHS) ⇒
    if |LHS| = 0 then Skip else S
  case SeqComp(S1,S2) ⇒
    if IsEmptyAssignment(S1) then
      RemoveEmptyAssignments(S2)
    else if IsEmptyAssignment(S2) then
      RemoveEmptyAssignments(S1)
    else
      SeqComp(RemoveEmptyAssignments(S1),
        RemoveEmptyAssignments(S2))
  case IF(B0,Sthen,Selse) ⇒
    IF(B0, RemoveEmptyAssignments(Sthen),
      RemoveEmptyAssignments(Selse))
  case DO(B,S1) ⇒
    DO(B, RemoveEmptyAssignments(S1))
  case Skip ⇒ Skip
  }
}

predicate IsEmptyAssignment(S: Statement)
  requires Valid(S) ∧ Core(S)
{
  IsAssignment(S) ∧ |GetLHS(S)| = 0
}

predicate NoEmptyAssignments(S: Statement)
  reads *
  requires Valid(S) ∧ Core(S)
{
  match S {
  case Assignment(LHS,RHS) ⇒
    LHS ≠ [] ∧ RHS ≠ []
  case SeqComp(S1,S2) ⇒
    NoEmptyAssignments(S1) ∧ NoEmptyAssignments(S2)
  case IF(B0,Sthen,Selse) ⇒
    NoEmptyAssignments(Sthen) ∧ NoEmptyAssignments(Selse)
  case DO(B,S1) ⇒
    NoEmptyAssignments(S1)
  case Skip ⇒ true
  }
}

predicate NoSelfAssignments(S: Statement)
  reads *
  requires Valid(S) ∧ Core(S)
{
  match S {
  case Assignment(LHS,RHS) ⇒
    NoSelfAssignmentsInAssignment(LHS, RHS)
  case SeqComp(S1,S2) ⇒
    NoSelfAssignments(S1) ∧ NoSelfAssignments(S2)
  case IF(B0,Sthen,Selse) ⇒
    NoSelfAssignments(Sthen) ∧ NoSelfAssignments(Selse)
  case DO(B,S1) ⇒
    NoSelfAssignments(S1)
  case Skip ⇒ true
  }
}

predicate NoSelfAssignmentsInAssignment(LHS: seq<Variable>,
  RHS: seq<Expression>)
  reads *
  requires Valid(Assignment(LHS, RHS))
{
  if LHS = [] then true
  else
    if LHS[0] = GetFirstVariableInRHS(RHS[0]) then false
    else NoSelfAssignmentsInAssignment(LHS[1..], RHS[1..])
}
```

A.2 Full proofs of theorems and lemmas

```
function RF(S': Statement, v': Variable): set<VarSlide>
  reads *
  requires Valid(S′) ∧ Core(S′)
{
  if v′ ∉ def(S′) then {}
  else
    var varSlides := VarSlideDGVarSlides(VarSlideDGOf(S′));
    var vSlide := VarSlideOfInstance(S′, v′);

    if VarSlideTag(vSlide) = Regular then {vSlide}
    else
      var instances := VarSlideInstances(vSlide, S′);
      (set i | i in instances ∧ i in def(S′) ∧
        VarSlideTag(VarSlideOfInstance(S′, i)) = Regular •
        VarSlideOfInstance(S′, i)) +
      (set i1, i2 | i1 in instances ∧ i1 in def(S′) ∧
        VarSlideTag(VarSlideOfInstance(S′, i1)) = Phi ∧
        i2 in RF(S′, i1) • i2)
}

function VarSlideOfInstance(S′: Statement, v′: Variable): VarSlide
  reads *
  requires Valid(S′) ∧ Core(S′)
  requires v′ in def(S′)
{
  var varSlideDG := VarSlideDGOf(S′);
  var varSlides := VarSlideDGVarSlides(varSlideDG);
  var vSlide :| vSlide in varSlides ∧ VarSlideVariable(vSlide) = v′;
  vSlide
}

function VarSlideInstances(vSlide: VarSlide, S′: Statement): set<Variable>
  reads *
  requires Valid(S′) ∧ Core(S′)
  requires vSlide in VarSlidesOf(S′, def(S′))
{
  (set l′, i | l′ in VarSlideLabels(vSlide, S′) ∧
    IsAssignment(slipOf(S′, l′)) ∧
    i in UsedVarsFor(S′, l′, VarSlideVariable(vSlide)) • i)
}

function statementSize(S: Statement): nat
  reads *
  requires Valid(S) ∧ Core(S)
  decreases S
{
  match S {
  case Skip ⇒ 1
  case Assignment(LHS, RHS) ⇒ 1
  case SeqComp(S1,S2) ⇒
    1 + statementSize(S1) + statementSize(S2)
  case IF(B0,Sthen,Selse) ⇒
    1 + statementSize(Sthen) + statementSize(Selse)
  case DO(B,Sloop) ⇒
    1 + statementSize(Sloop)
  }
}
```

A.2.1 Reaching Definitions and Liveness for exit

```
lemma ReachingAndLivenessForExit(S: Statement, S′: Statement, v: Variable,
  l: Label, v′: Variable, l′: Label, XLs: seq<set<Variable>>,
  X: seq<Variable>, V′: set<Variable>)
  requires Valid(S) ∧ Valid(S′)
  requires Core(S) ∧ Core(S')
  requires ValidLabel(l, S)
  requires ValidLabel(l', S')
  requires ValidXLs(glob(S), XLs, X)
  requires RemoveEmptyAssignments(Rename(S', XLs, X, glob(S))) = S
  requires v' in LiveOnExit(S', V', l')
  requires l' = VarLabelOf(S, S', l, XLs, X)
  requires MatchingSlipsToSSA(S, l, S', l')
  requires v' in InstancesOf(S', v, X, XLs, glob(S))
  requires OneLiveInstance(S, S', v, l, v', l', XLs, X, V')
  ensures varSlidesOfRDOUT(S, S', v, l, XLs, X) = RF(S', v')
  decreases S, distanceFromEntryForExit(S, l)
{
  if v ∉ def(slipOf(S, l)) {
    calc {
      RF(S', v');
    = { assert v' in LiveOnEntryFor(S', V', l') by {
          RLExitL1(S, S', V', l, l', v, v', XLs, X); }
        ReachingAndLivenessForEntry(S, S', v, l, v', l', XLs, X, V'); }
      varSlidesOfRDIN(S, S', v, l, XLs, X);
    = { assert ReachingDefinitionsInFor(S, l, v) =
          ReachingDefinitionsOutFor(S, l, v) by { RLExitL2(S, l, v); } }
      varSlidesOfRDOUT(S, S', v, l, XLs, X);
    }
  }
  else {
    match slipOf(S, l) {
    case Skip ⇒  assert false;
    case Assignment(LHS, RHS) ⇒
      assert {(v, l)} = ReachingDefinitionsOutFor(S, l, v) by {
        RLExitL3(S, l, v); }
      var slide := (l, v);
      var varSlide := VarSlideOf(S, S', slide, XLs, X);
      assert {varSlide} = RF(S', v') by {
        RLExitL4(S, S', l, l', varSlide, v, v', XLs, X); }
    case SeqComp(S1,S2) ⇒
      calc {
        RF(S', v');
      = { assert v' in LiveOnExit(S', V', l'+[2]) by {
            RLExitL5(S, S', V', l, l', v', XLs, X); }
          ReachingAndLivenessForExit(S, S', v, l+[2],
            v', l'+[2], XLs, X, V'); }
        varSlidesOfRDOUT(S, S', v, l+[2], XLs, X);
      = { RLExitL6(S, S', l, v, XLs, X); }
        varSlidesOfRDOUT(S, S', v, l, XLs, X);
      }
    case IF(B0,Sthen,Selse) ⇒
      var ThenPhiLabel := l'+[1,2];
      var vT' :| vT' in GetCorrespondingExpression(
        GetLHS(slipOf(S', ThenPhiLabel)),
        GetRHS(slipOf(S', ThenPhiLabel)), v').1;
      var ElsePhiLabel := l'+[2,2];
      var vE' :| vE' in GetCorrespondingExpression(
        GetLHS(slipOf(S', ElsePhiLabel)),
        GetRHS(slipOf(S', ElsePhiLabel)), v').1;

      calc {
        RF(S', v');
      = { RLExitL7(S', V', l', v', ThenPhiLabel, vT',
            ElsePhiLabel, vE'); }
          RF(S', vT') + RF(S', vE');
      = { assert vT' in LiveOnExit(S', V', l'+[1,1]) by {
            RLExitL8(S', V', l', v', vT'); }
          ReachingAndLivenessForExit(S, S', v, l+[1],
            vT', l'+[1,1], XLs, X, V');
          assert varSlidesOfRDOUT(S, S', v, l+[1], XLs, X) =
            RF(S', vT'); }
      varSlidesOfRDOUT(S, S', v, l+[1], XLs, X) + RF(S', vE');
    = { assert vE' in LiveOnExit(S', V', l'+[2,1]) by {
            RLExitL9(S', V', l', v', vE'); }
          ReachingAndLivenessForExit(S, S', v, l+[2],
            vE', l'+[2,1], XLs, X, V');
          assert varSlidesOfRDOUT(S, S', v, l+[2], XLs, X) =
            RF(S', vE'); }
      varSlidesOfRDOUT(S, S', v, l+[1], XLs, X) +
        varSlidesOfRDOUT(S, S', v, l+[2], XLs, X);
    = { RLExitL10(S, S', l, v, v', XLs, X); }
      varSlidesOfRDOUT(S, S', v, l, XLs, X);
    }
  case DO(B,Sloop) ⇒
    var InitPhiLabel := l'+[1];
    var vI' :| vI' in GetCorrespondingExpression(
      GetLHS(slipOf(S', InitPhiLabel)),
      GetRHS(slipOf(S', InitPhiLabel)), v').1;
    var BodyPhiLabel := l'+[2,1,2];
    var vB' :| vB' in GetCorrespondingExpression(
      GetLHS(slipOf(S', BodyPhiLabel)),
      GetRHS(slipOf(S', BodyPhiLabel)), v').1;

    calc {
      RF(S', v');
    = { RLExitL11(S', V', l', v', InitPhiLabel, vI',
            BodyPhiLabel, vB'); }
      RF(S', vI') + RF(S', vB');
    = { assert vI' in LiveOnEntryFor(S', V', l') by {
            RLExitL12(S', V', l', v', vI'); }
          ReachingAndLivenessForEntry(S, S', v, l, vI', l', XLs, X, V');
          assert varSlidesOfRDIN(S, S', v, l, XLs, X) = RF(S', vI'); }
      varSlidesOfRDIN(S, S', v, l, XLs, X) + RF(S', vB');
    = { assert vB' in LiveOnExit(S', V', l'+[2,1,1]) by {
            RLExitL13(S', V', l', v', vB'); }
          ReachingAndLivenessForExit(S, S', v, l+[1],
            vB', l'+[2,1,1], XLs, X, V');
          assert varSlidesOfRDOUT(S, S', v, l+[1], XLs, X) =
            RF(S', vB'); }
      varSlidesOfRDIN(S, S', v, l, XLs, X) +
        varSlidesOfRDOUT(S, S', v, l+[1], XLs, X);
    = { RLExitL14(S, S', V', l, l', v, v', XLs, X); }
      varSlidesOfRDOUT(S, S', v, l, XLs, X);
    }
  }
 }
}

function varSlidesOfRDOUT(S: Statement, S': Statement, v: Variable,
  l: Label, XLs: seq<set<Variable>>, X: seq<Variable>): set<VarSlide>
  reads *
  requires Valid(S) ∧ Valid(S')
  requires Core(S) ∧ Core(S')
  requires ValidXLs(glob(S), XLs, X)
  requires ValidLabel(l, S)
  requires S = RemoveEmptyAssignments(Rename(S', XLs, X, glob(S)))
{
  (set pair | pair in ReachingDefinitionsOutFor(S, l, v) •
    var slide := (pair.1, pair.0); VarSlideOf(S, S', slide, XLs, X))
}
```

```
function distanceFromEntryForExit(S: Statement, l: Label): nat
  reads *
  requires Valid(S) ∧ Core(S) ∧ ValidLabel(l, S)
  decreases S
{
  if l = [] then 2*statementSize(S)
  else
    assert ¬IsSkip(S) ∧ ¬IsAssignment(S);
    match S {
    case SeqComp(S1,S2) ⇒
      if l[0] = 1 then 1 + distanceFromEntryForExit(S1, l[1..])
      else 1 + 2*statementSize(S1) + distanceFromEntryForExit(S2, l[1..])
    case IF(B0,Sthen,Selse) ⇒
      if l[0] = 1 then 1 + distanceFromEntryForExit(Sthen, l[1..]) else
      1 + 2*statementSize(Sthen) + distanceFromEntryForExit(Selse, l[1..])
    case DO(B,Sloop) ⇒
      1 + distanceFromEntryForExit(Sloop, l[1..])
    }
}

predicate OneLiveInstance(S: Statement, S': Statement, v: Variable,
  l: Label, v': Variable, l': Label, XLs: seq<set<Variable>>,
  X: seq<Variable>, V': set<Variable>)
  reads *
  requires Valid(S) ∧ Valid(S')
  requires Core(S) ∧ Core(S')
  requires ValidLabel(l, S)
  requires ValidLabel(l', S')
  requires ValidXLs(glob(S), XLs, X)
  requires RemoveEmptyAssignments(Rename(S', XLs, X, glob(S))) = S
  requires l' = VarLabelOf(S, S', l, XLs, X)
  requires v' in InstancesOf(S', v, X, XLs, glob(S))
  requires v' in LiveOnExit(S', V', l')
{
  ∀ u' • u' in InstancesOf(S', v, X, XLs, glob(S)) ∧
  u' ≠ v' ∧ IsAssignment(slipOf(S', l')) ⟹ u' ∉ GetLHS(slipOf(S', l'))
}
```

A.2.2 Reaching Definitions and Liveness for entry

```
lemma ReachingAndLivenessForEntry(S: Statement, S': Statement,
  v: Variable, l: Label, v': Variable, l': Label,
  XLs: seq<set<Variable>>, X: seq<Variable>, V': set<Variable>)
  requires Valid(S) ∧ Valid(S')
  requires Core(S) ∧ Core(S')
  requires ValidLabel(l, S)
  requires ValidLabel(l', S')
  requires ValidXLs(glob(S), XLs, X)
  requires RemoveEmptyAssignments(Rename(S', XLs, X, glob(S))) = S
  requires v' in LiveOnEntryFor(S', V', l')
  requires l' = VarLabelOf(S, S', l, XLs, X)
  requires MatchingSlipsToSSA(S, l, S', l')
  requires v' in InstancesOf(S', v, X, XLs, glob(S))
  ensures varSlidesOfRDIN(S, S', v, l, XLs, X) = RF(S', v')
  decreases S, distanceFromEntryForEntry(S, l)
{
  if l = [] {
    var emptySet := {};
    calc {
      RF(S', v');
    = { assert v' ∉ def(S') by { RLEntryL1(S', V', v'); } }
```

```
      emptySet;
    = { assert ReachingDefinitionsInFor(S, [], v) = {}; }
      varSlidesOfRDIN(S, S', v, l, XLs, X);
    }
  }
  else {
    var l1, c :| l1 + [c] = l;
    match slipOf(S, l1) {
    case SeqComp(S1,S2) ⇒
      var l1' := VarLabelOf(S, S', l1, XLs, X);
      assert l1' + [c] = l';
      assert IsSeqComp(slipOf(S', l1'));
      if c = 1 {
        calc {
          RF(S', v');
        = { assert v' in LiveOnEntryFor(S', V', l1') by {
                RLEntryL2(S', V', l1', l', v'); }
              ReachingAndLivenessForEntry(S, S', v, l1,
                v', l1', XLs, X, V'); }
          varSlidesOfRDIN(S, S', v, l1, XLs, X);
        = { RLEntryL3(S, S', l, l1, v, XLs, X); }
          varSlidesOfRDIN(S, S', v, l, XLs, X);
        }
      }
      else {
        assert c = 2;
        calc {
          RF(S', v');
        = { assert v' in LiveOnExit(S', V', l1'+[1]) by {
                RLEntryL4(S', V', l1', l', v'); }
              ReachingAndLivenessForExit(S, S', v, l1+[1],
                v', l1'+[1], XLs, X, V'); }
          varSlidesOfRDOUT(S, S', v, l1+[1], XLs, X);
        = { RLEntryL5(S, S', l, l1, v, XLs, X); }
          varSlidesOfRDIN(S, S', v, l1+[2], XLs, X);
        = { assert l = l1 + [2]; }
          varSlidesOfRDIN(S, S', v, l, XLs, X);
        }
      }
    case IF(B0,Sthen,Selse) ⇒
      var l1' := VarLabelOf(S, S', l1, XLs, X);
      assert l1' + [c,1] = l';
      assert IsIF(slipOf(S', l1'));
      calc {
        RF(S', v');
      = { assert v' in LiveOnEntryFor(S', V', l1') by {
              RLEntryL6(S', V', l1', l', v'); }
            ReachingAndLivenessForEntry(S, S', v, l1,
              v', l1', XLs, X, V'); }
        varSlidesOfRDIN(S, S', v, l1, XLs, X);
      = { RLEntryL7(S, S', l, l1, v, XLs, X); }
        varSlidesOfRDIN(S, S', v, l, XLs, X);
      }
    case DO(B,Sloop) ⇒
      var l1' := VarLabelOf(S, S', l1, XLs, X);
      assert l1' + [2,1,1] = l';
      assert IsSeqComp(slipOf(S', l1'));

      if (v in def(slipOf(S, l))) {
        var Sloop' := slipOf(S', l');
        var InitPhiLabel := l1'+[1];
        var vI' :| vI' in GetCorrespondingExpression(
          GetLHS(slipOf(S', InitPhiLabel)),
          GetRHS(slipOf(S', InitPhiLabel)), v').1;
        var BodyPhiLabel := l1'+[2,1,2];
        var vB' :| vB' in GetCorrespondingExpression(
          GetLHS(slipOf(S', BodyPhiLabel)),
          GetRHS(slipOf(S', BodyPhiLabel)), v').1;

        calc {
          RF(S', v');
        = { RLEntryL8(S', V', l1', v',
              InitPhiLabel, vI', BodyPhiLabel, vB'); }
          RF(S', vI') + RF(S', vB');
        = { assert vI' in LiveOnEntryFor(S', V', l1') by {
              RLEntryL9(S', V', l1', l', v', vI'); }
            ReachingAndLivenessForEntry(S, S', v, l1,
              vI', l1', XLs, X, V');
            assert varSlidesOfRDIN(S, S', v, l1, XLs, X) =
              RF(S', vI'); }
          varSlidesOfRDIN(S, S', v, l1, XLs, X) + RF(S', vB');
        = { assert vB' in LiveOnExit(S', V', l') by {
              RLEntryL10(S', V', l1', l', v', vB'); }
            assert l1' + [2,1,1] = l';
            ReachingAndLivenessForExit(Sloop, Sloop', v, [],
              vB', [], XLs, X, {vB'});
            assert varSlidesOfRDOUT(Sloop, Sloop', v, [], XLs, X) =
              RF(Sloop', vB');
            assert RF(S', vI') + RF(Sloop', vB') = RF(S', vB'); }
          varSlidesOfRDIN(S, S', v, l1, XLs, X) +
            varSlidesOfRDOUT(Sloop, Sloop', v, [], XLs, X);
        =  { RLEntryL11(S, S', Sloop, Sloop', V', l1, l, v, XLs, X); }
          varSlidesOfRDIN(S, S', v, l, XLs, X);
        }
      }
      else {
        calc {
          RF(S', v');
        = { assert v' in LiveOnEntryFor(S', V', l1') by {
              RLEntryL12(S', V', l1', l', v'); }
            ReachingAndLivenessForEntry(S, S', v, l1,
              v', l1', XLs, X, V'); }
          varSlidesOfRDIN(S, S', v, l1, XLs, X);
        = { RLEntryL13(S, S', l, l1, v, XLs, X); }
          varSlidesOfRDIN(S, S', v, l, XLs, X);
        }
      }
    }
  }
}

function varSlidesOfRDIN(S: Statement, S': Statement, v: Variable,
  l: Label, XLs: seq<set<Variable>>, X: seq<Variable>): set<VarSlide>
  reads *
  requires Valid(S) ∧ Valid(S')
  requires Core(S) ∧ Core(S')
  requires ValidXLs(glob(S), XLs, X)
  requires ValidLabel(l, S)
  requires S = RemoveEmptyAssignments(Rename(S', XLs, X, glob(S)))
{
  (set pair | pair in ReachingDefinitionsInFor(S, l, v) •
    var slide := (pair.1, pair.0); VarSlideOf(S, S', slide, XLs, X))
}

function distanceFromEntryForEntry(S: Statement, l: Label): nat
  reads *
  requires Valid(S) ∧ Core(S) ∧ ValidLabel(l, S)
  decreases S
{
  if l = [] then 1
  else
    assert ¬IsSkip(S) ∧ ¬IsAssignment(S);
    match S {
    case SeqComp(S1,S2) ⇒
      if l[0] = 1 then 1 + distanceFromEntryForEntry(S1, l[1..])
      else 1 + 2*statementSize(S1) + distanceFromEntryForEntry(S2, l[1..])
    case IF(B0,Sthen,Selse) ⇒
      if l[0] = 1 then 1 + distanceFromEntryForEntry(Sthen, l[1..])
      else 1 + 2*statementSize(Sthen) +
        distanceFromEntryForEntry(Selse, l[1..])
    case DO(B,Sloop) ⇒
      1 + distanceFromEntryForEntry(Sloop, l[1..])
    }
}
```

A.2.3 Identical Skip Slips

Lemma 11 (LemmaIdenticalSkipSlips).

```
lemma LemmaIdenticalSkipSlips(S: Statement, S': Statement,
  V: set<Variable>, SV: Statement, res: Statement, SV': Statement,
  V': set<Variable>, XLs: seq<set<Variable>>, X: seq<Variable>,
  l: Label, slidesSV: set<Slide>, varSlidesSV: set<VarSlide>)
  requires Valid(S) ∧ Valid(SV) ∧ Valid(SV') ∧ Valid(res)
  requires Core(S) ∧ Core(SV) ∧ Core(SV') ∧ Core(res)
  requires SliceOf(S,V).1 = SV
  requires RemoveEmptyAssignments(Rename(SV', XLs, X, glob(res))) = res
  requires RemoveEmptyAssignments(Rename(S', XLs, X, glob(res))) = S
  requires ∀ Sm • Sm in slidesSV ⟺ (Sm in SlidesOf(S, def(S))
    ∧ ∃ Sn • Sn in FinalDefSlides(S, V) ∧
    SlideDGReachable(SlideDGOf(S), Sm, Sn, SlideDGSlides(SlideDGOf(S))))
  requires ∀ vSlide • vSlide in varSlidesSV ⟹
    (vSlide in VarSlidesOf(S', def(S)) ∧
    ∃ Sn: VarSlide • VarSlideVariable(Sn) in V' ∧
    VarSlideDGReachable(VarSlideDGOf(S'), vSlide,
      Sn, VarSlideDGVarSlides(VarSlideDGOf(S'))))
  requires ∀ slide • slide in SlideDGSlides(SlideDGOf(S)) ⟹
    (slide in slidesSV ⟺
      VarSlideOf(S, S', slide, XLs, X) in varSlidesSV)
  requires ValidLabel(l, S) ∧ ValidLabel(l, SV) ∧ ValidLabel(l, res)
  requires MatchingSlips(S, res, l)
  ensures IsSkip(slipOf(SV, l)) ⟺ IsSkip(slipOf(res, l))
{
  assert IsSkip(slipOf(SV, l)) ⟺ IsSkip(slipOf(res, l)) by {
    if (IsSkip(slipOf(SV, l)))
    {
      LemmaIdenticalSkipSlipsA(S, S', V, SV, res, SV',
        V', XLs, X, l, slidesSV, varSlidesSV);
    }
    else
    {
      LemmaIdenticalSkipSlipsB(S, S', V, SV, res, SV',
        V', XLs, X, l, slidesSV, varSlidesSV);
    }

}

lemma LemmaIdenticalSkipSlipsA(S: Statement, S': Statement,
  V: set<Variable>, SV: Statement, res: Statement, SV': Statement,
  V': set<Variable>, XLs: seq<set<Variable>>, X: seq<Variable>,
  l: Label, slidesSV: set<Slide>, varSlidesSV: set<VarSlide>)
  requires Valid(S) ∧ Valid(SV) ∧ Valid(SV') ∧ Valid(res)
  requires Core(S) ∧ Core(SV) ∧ Core(SV') ∧ Core(res)
  requires SliceOf(S,V).1 = SV
  requires RemoveEmptyAssignments(Rename(SV', XLs, X, glob(res))) = res
  requires RemoveEmptyAssignments(Rename(S', XLs, X, glob(res))) = S
  requires ∀ Sm • Sm in slidesSV ⟺ (Sm in SlidesOf(S, def(S))
    ∧ ∃ Sn • Sn in FinalDefSlides(S, V) ∧
    SlideDGReachable(SlideDGOf(S), Sm, Sn, SlideDGSlides(SlideDGOf(S))))
  requires ∀ vSlide • vSlide in varSlidesSV ⟹
    (vSlide in VarSlidesOf(S', def(S)) ∧
    ∃ Sn: VarSlide • VarSlideVariable(Sn) in V' ∧
    VarSlideDGReachable(VarSlideDGOf(S'), vSlide,
      Sn, VarSlideDGVarSlides(VarSlideDGOf(S'))))
  requires ∀ slide • slide in SlideDGSlides(SlideDGOf(S)) ⟹
    (slide in slidesSV ⟺
      VarSlideOf(S, S', slide, XLs, X) in varSlidesSV)
  requires ValidLabel(l, S) ∧ ValidLabel(l, SV) ∧ ValidLabel(l, res)
  requires MatchingSlips(S, res, l)
  requires IsSkip(slipOf(SV, l))
  ensures IsSkip(slipOf(res, l))
{
  var l' := VarLabelOf(S, SV', l, XLs, X);
  var slidesOfSlipSV := PrefixOfSlideLabel(slidesSV, l);
  var varSlidesOfSlipSV' := PrefixOfVarSlideLabel(varSlidesSV, l', SV');

  assert IsSkip(slipOf(res, l)) by {
    calc {
      IsSkip(slipOf(SV, l));
    ⟹ { assert slidesOfSlipSV = {} ⟺ IsSkip(slipOf(SV, l)); }
      slidesOfSlipSV = {};
    ⟹ { assert |slidesOfSlipSV| = |varSlidesOfSlipSV'|; }
      varSlidesOfSlipSV' = {};
    ⟹ { assert varSlidesOfSlipSV' = {} ⟺ IsSkip(slipOf(SV', l')); }
      IsSkip(slipOf(SV', l'));
    ⟹ { assert MatchingSlipsFromSSA(SV', l', res, l) by {
            LemmaMatchingSlipsFromSSA(SV', l', res, l, XLs, X); } }
      IsSkip(slipOf(res, l));
    }
  }
}

lemma LemmaIdenticalSkipSlipsB(S: Statement, S': Statement,
  V: set<Variable>, SV: Statement, res: Statement, SV': Statement,
  V': set<Variable>, XLs: seq<set<Variable>>, X: seq<Variable>,
  l: Label, slidesSV: set<Slide>, varSlidesSV: set<VarSlide>)
  requires Valid(S) ∧ Valid(SV) ∧ Valid(SV') ∧ Valid(res)
  requires Core(S) ∧ Core(SV) ∧ Core(SV') ∧ Core(res)
  requires SliceOf(S,V).1 = SV
  requires RemoveEmptyAssignments(Rename(SV', XLs, X, glob(res))) = res
  requires RemoveEmptyAssignments(Rename(S', XLs, X, glob(res))) = S
  requires ∀ Sm • Sm in slidesSV ⟺ (Sm in SlidesOf(S, def(S))
    ∧ ∃ Sn • Sn in FinalDefSlides(S, V) ∧
    SlideDGReachable(SlideDGOf(S), Sm, Sn, SlideDGSlides(SlideDGOf(S))))
  requires ∀ vSlide • vSlide in varSlidesSV ⟹
    (vSlide in VarSlidesOf(S', def(S)) ∧
    ∃ Sn: VarSlide • VarSlideVariable(Sn) in V' ∧
    VarSlideDGReachable(VarSlideDGOf(S'), vSlide,
      Sn, VarSlideDGVarSlides(VarSlideDGOf(S'))))
  requires ∀ slide • slide in SlideDGSlides(SlideDGOf(S)) ⟹
    (slide in slidesSV ⟺
    VarSlideOf(S, S', slide, XLs, X) in varSlidesSV)
  requires ValidLabel(l, S) ∧ ValidLabel(l, SV) ∧ ValidLabel(l, res)
  requires MatchingSlips(S, res, l)
  requires ¬IsSkip(slipOf(SV, l))
  ensures ¬IsSkip(slipOf(res, l))
{
  var l' := VarLabelOf(S, SV', l, XLs, X);
  var slidesOfSlipSV := PrefixOfSlideLabel(slidesSV, l);
  var varSlidesOfSlipSV' := PrefixOfVarSlideLabel(varSlidesSV, l', SV');

  assert ¬IsSkip(slipOf(res, l)) by {
    calc {
      ¬IsSkip(slipOf(SV, l));
    ⟹ { assert slidesOfSlipSV = {} ⟺ IsSkip(slipOf(SV, l)); }}
      slidesOfSlipSV ≠ {};
    ⟹ { assert |slidesOfSlipSV| = |varSlidesOfSlipSV'|; }
      varSlidesOfSlipSV' ≠ {};
    ⟹ { assert varSlidesOfSlipSV' = {} ⟺ IsSkip(slipOf(SV', l')); }
      ¬IsSkip(slipOf(SV', l'));
    ⟹ { LemmaRenameSkip(slipOf(SV', l'), XLs
          X, glob(slipOf(res, l))); }
      ¬IsSkip(Rename(slipOf(SV', l'), XLs, X, glob(slipOf(res, l))));
    ⟹ { LemmaRemoveEmptyAssignmentsSkip(Rename(slipOf(SV', l'),
          XLs, X, glob(slipOf(res, l)))); }
      ¬IsSkip(RemoveEmptyAssignments(Rename(slipOf(SV', l'), XLs,
        X, glob(slipOf(res, l)))));
    = { LemmaCommutativeFromSSASlips(SV', XLs, X, l', l); }
      ¬IsSkip(slipOf(RemoveEmptyAssignments(Rename(SV', XLs,
        X, glob(res))), l));
    ⟹ { assert RemoveEmptyAssignments(Rename(SV', XLs,
          X, glob(res))) = res; }
      ¬IsSkip(slipOf(res, l));
    }
  }
}
```

A.2.4 Inverse varLabelOf

Lemma 12 (LemmaInverseVarLabelOf).

```
function LabelOf(S': Statement, S: Statement, l': Label,
  XLs: seq<set<Variable>>, X: seq<Variable>): Label
  reads *
  requires Valid(S) ∧ Valid(S')
  requires Core(S) ∧ Core(S')
  requires ValidLabel(l', S')
  requires ValidXLs(glob(S), XLs, X)
  requires S = RemoveEmptyAssignments(Rename(S', XLs, X, glob(S)))
  requires MatchingSlipsFromSSA(S', [], S, [])
  ensures ValidLabel(LabelOf(S', S, l', XLs, X), S)
{
  match S' {
  case Skip ⇒
    assert IsSkip(S);
    assert l' = [];
    []
  case Assignment(LHS',RHS') ⇒
    assert IsAssignment(S) ∨ IsSkip(S);
    assert l' = [];
    []
  case SeqComp(S1',S2') ⇒
```

```
    if l' = [] then []
    else
      assert IsSeqComp(S) ∨ IsDO(S);
      if IsSeqComp(S) then
        match S {
        case SeqComp(S1,S2) ⇒
          if l'[0] = 1 then [1] + LabelOf(S1', S1, l'[1..], XLs, X)
          else [2] + LabelOf(S2', S2, l'[1..], XLs, X)
        }
      else
        assert IsDO(S2') ∧ IsSeqComp(GetLoopBody(S2'));
        var Sloop' := GetS1(GetLoopBody(S2'));
        match S {
        case DO(B,Sloop) ⇒ [1] + LabelOf(Sloop', Sloop, l'[3..], XLs, X)
        }
  case IF(B0',Sthen',Selse') ⇒
    assert IsIF(S);
    if l' = [] then [] else
      match S {
      case IF(B0,Sthen,Selse) ⇒
        if l'[0] = 1 then [1] + LabelOf(Sthen', Sthen, l'[2..], XLs, X)
        else [2] + LabelOf(Selse', Selse, l'[2..], XLs, X)
      }
  }
}

predicate InverseVarLabelOf(S: Statement, S': Statement, SV': Statement,
  res: Statement, XLs: seq<set<Variable>>, X: seq<Variable>)
  reads *
  requires Valid(S) ∧ Valid(S') ∧ Valid(SV') ∧ Valid(res)
  requires Core(S) ∧ Core(S') ∧ Core(SV') ∧ Core(res)
  requires ValidXLs(glob(S), XLs, X)
  requires ValidXLs(glob(res), XLs, X)
  requires S = RemoveEmptyAssignments(Rename(S', XLs, X, glob(S)))
  requires MatchingSlipsToSSA(S, [], S', [])
  requires res = RemoveEmptyAssignments(Rename(SV', XLs, X, glob(res)))
  requires MatchingSlipsFromSSA(SV', [], res, [])
{
  ∀ l, l' • ValidLabel(l, S) ∧ l' = VarLabelOf(S, S', l, XLs, X) ∧
    ValidLabel(l', S') ∧ ValidLabel(l', SV') ∧ NoSelfAssignments(S) ∧
    NoEmptyAssignments(S) ⟹ l = LabelOf(SV', res, l', XLs, X)
}

lemma LemmaInverseVarLabelOf(S: Statement, S': Statement, SV': Statement,
  res: Statement, XLs: seq<set<Variable>>, X: seq<Variable>)
  requires Valid(S) ∧ Valid(S') ∧ Valid(SV') ∧ Valid(res)
  requires Core(S) ∧ Core(S') ∧ Core(SV') ∧ Core(res)
  requires ValidXLs(glob(S), XLs, X)
  requires ValidXLs(glob(res), XLs, X)
  requires S = RemoveEmptyAssignments(Rename(S', XLs, X, glob(S)))
  requires MatchingSlipsToSSA(S, [], S', [])
  requires res = RemoveEmptyAssignments(Rename(SV', XLs, X, glob(res)))
  requires MatchingSlipsFromSSA(SV', [], res, [])
  ensures InverseVarLabelOf(S, S', SV', res, XLs, X)
{
  ∀ l, l' | ValidLabel(l, S) ∧ l' = VarLabelOf(S, S', l, XLs, X) ∧
    ValidLabel(l', S') ∧ ValidLabel(l', SV') ∧ NoSelfAssignments(S) ∧
    NoEmptyAssignments(S) ensures l = LabelOf(SV', res, l', XLs, X) {
      LemmaInverseVarLabelOfRec(S, S', SV', res, l, l', XLs, X);
    }
}

lemma LemmaInverseVarLabelOfRec(S: Statement, S': Statement,
  SV': Statement, res: Statement, l: Label, l': Label,
  XLs: seq<set<Variable>>, X: seq<Variable>)
  requires Valid(S) ∧ Valid(S') ∧ Valid(SV') ∧ Valid(res)
  requires Core(S) ∧ Core(S') ∧ Core(SV') ∧ Core(res)
  requires ValidXLs(glob(S), XLs, X)
  requires ValidXLs(glob(res), XLs, X)
  requires S = RemoveEmptyAssignments(Rename(S', XLs, X, glob(S)))
  requires MatchingSlipsToSSA(S, [], S', [])
  requires res = RemoveEmptyAssignments(Rename(SV', XLs, X, glob(res)))
  requires MatchingSlipsFromSSA(SV', [], res, [])
  requires ValidLabel(l, S) ∧ ValidLabel(l', S') ∧ ValidLabel(l', SV')
  requires l' = VarLabelOf(S, S', l, XLs, X)
  ensures l = LabelOf(SV', res, l', XLs, X)
{
  match SV' {
  case Skip ⇒
    assert IsSkip(res);
    assert l' = [];
    assert l = [];
  case Assignment(LHS',RHS') ⇒
    assert IsAssignment(res) ∨ IsSkip(res);
    assert l' = [];
    assert l = [];
  case SeqComp(S1',S2') ⇒
    if l' = [] { assert l = []; } else {
      assert IsSeqComp(res) ∨ IsDO(res);
      if IsSeqComp(res) {
        var S1, S2 := GetS1(res), GetS2(res);
        if l'[0] = 1 {
          calc {
            LabelOf(SV', res, l', XLs, X);
          = { assert IsSeqComp(SV') ∧ IsSeqComp(res) ∧ l'[0] = 1; }
            [1] + LabelOf(S1', S1, l'[1..], XLs, X);
          = { LemmaInverseVarLabelOfRec(GetS1(S), GetS1(S'), S1', S1,
                  l[1..], l'[1..], XLs, X);
                assert l[1..] = LabelOf(S1', S1, l'[1..], XLs, X); }
            [1] + l[1..];
          = { assert l[0] = 1 by {
                  assert l' = VarLabelOf(S, S', l, XLs, X); } }
            l;
          }
        }
        else {
          calc {
            LabelOf(SV', res, l', XLs, X);
          = { assert IsSeqComp(SV') ∧ IsSeqComp(res) ∧ l'[0] = 2; }
            [2] + LabelOf(S2', S2, l'[1..], XLs, X);
          = { LemmaInverseVarLabelOfRec(GetS2(S), GetS2(S'), S2', S2,
                  l[1..], l'[1..], XLs, X);
                assert l[1..] = LabelOf(S2', S2, l'[1..], XLs, X); }
            [2] + l[1..];
          = { assert l[0] = 2 by {
                  assert l' = VarLabelOf(S, S', l, XLs, X); } }
            l;
          }
        }
      }
      else {
        var Sloop' := GetS1(GetLoopBody(S2'));
        var B, Sloop := GetLoopBool(res), GetLoopBody(res);
        calc {
          LabelOf(SV', res, l', XLs, X);
        = { assert IsSeqComp(SV') ∧ IsDO(res); }
```

```
                [1] + LabelOf(Sloop', Sloop, l'[3..], XLs, X);
        = { LemmaInverseVarLabelOfRec(GetLoopBody(S),
                GetS1(GetLoopBody(GetS2(S'))), Sloop', Sloop, l[1..],
                l'[3..], XLs, X);
              assert l[1..] = LabelOf(Sloop', Sloop, l'[3..], XLs, X); }
          [1] + l[1..];
        = { assert l[0] = 1; }
          l;
        }
      }
    }
  case IF(B0',Sthen',Selse') ⇒
    assert IsIF(res);
    if l' = [] { assert l = []; } else {
      var B0, Sthen, Selse :=
        GetIfBool(res), GetIfThen(res), GetIfElse(res);
      if l'[0] = 1 {
        calc {
          LabelOf(SV', res, l', XLs, X);
        = { assert IsIF(SV') ∧ IsIF(res) ∧ l'[0] = 1; }
          [1] + LabelOf(Sthen', Sthen, l'[2..], XLs, X);
        = { LemmaInverseVarLabelOfRec(GetIfThen(S), GetIfThen(S'),
                Sthen', Sthen, l[1..], l'[2..], XLs, X);
              assert l[1..] = LabelOf(Sthen', Sthen, l'[2..], XLs, X); }
          [1] + l[1..];
        = { assert l[0] = 1 by {
                assert l' = VarLabelOf(S, S', l, XLs, X); } }
          l;
        }
      }
      else {
        calc {
          LabelOf(SV', res, l', XLs, X);
        = { assert IsIF(SV') ∧ IsIF(res) ∧ l'[0] = 2; }
          [2] + LabelOf(Selse', Selse, l'[2..], XLs, X);
        = { LemmaInverseVarLabelOfRec(GetIfElse(S), GetIfElse(S'),
                Selse', Selse, l[1..], l'[2..], XLs, X);
              assert l[1..] = LabelOf(Selse', Selse, l'[2..], XLs, X); }
          [2] + l[1..];
        = { assert l[0] = 2 by {
                assert l' = VarLabelOf(S, S', l, XLs, X); } }
          l;
        }
      }
    }
  }
}
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[2] Afshar Alam and Tendai Padenga. *Application software reengineering*. Pearson Education, 2010.

[3] Thomas Ball and Susan Horwitz. "Slicing Programs with Arbitrary Control-flow". In: *Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93, Linköping, Sweden, May 3-5, 1993, Proceedings*. Ed. by Peter Fritzson. Vol. 749. Lecture Notes in Computer Science. Springer, 1993, pp. 206–222.

[4] Chen Cozocaru. "The slide dependence graph and its use in software evolution". MA thesis. The Open University of Israel, 2014.

[5] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490.

[6] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Commun. ACM* 18.8 (1975), pp. 453–457.

[7] Ran Ettinger. "Refactoring via program slicing and sliding". PhD thesis. University of Oxford, UK, 2006.

[8] Ran Ettinger, Shmuel S. Tyszberowicz, and Shay Menaia. "Efficient method extraction for automatic elimination of type-3 clones". In: *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. 2017, pp. 327–337.

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), pp. 319–349.

[10] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999.

[11] Penny A. Grubb and Armstrong A. Takang. *Software maintenance - concepts and practice (2. ed.)* World Scientific, 2003. ISBN: 978-981-238-426-3.

[12] Mark Harman, David W. Binkley, and Sebastian Danicic. "Amorphous program slicing". In: *Journal of Systems and Software* 68.1 (2003), pp. 45–64.

[13] Mark Harman and Robert Hierons. "An Overview of Program Slicing". In: *Software Focus* 2 (Dec. 2001).

[14] Bogdan Korel and Janusz W. Laski. "Dynamic Program Slicing". In: *Inf. Process. Lett.* 29.3 (1988), pp. 155–163.

[15] Meir M. Lehman. "On understanding laws, evolution, and conservation in the large-program life cycle". In: *Journal of Systems and Software* 1 (1980), pp. 213–221.

[16] K. Rustan M. Leino. “Developing verified programs with dafny”. In: *35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by David Notkin, Betty H. C. Cheng, and Klaus Pohl. IEEE, 2013, pp. 1488–1490. DOI: 10.1109/ICSE.2013.6606754.

[17] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[18] Liang Tan and Christoph Bockisch. “A Survey of Refactoring Detection Tools”. In: *Workshops of the Software Engineering Conference*. Ed. by Stephan Krusche et al. Vol. 2308. CEUR-WS.org, 2019, pp. 100–105.

[19] Frank Tip. “A survey of program slicing techniques”. In: *J. Prog. Lang.* 3.3 (1995).

[20] Mark Weiser. “Program Slicing”. In: *ICSE*. IEEE Computer Society, 1981, pp. 439–449.

[21] Mark Weiser. “Programmers Use Slices When Debugging”. In: *Commun. ACM* 25.7 (1982), pp. 446–452.

[22] Wolfgang Wögerer and Technische Universität Wien. *A Survey of Static Program Analysis Techniques*. 2005.