Refactoring via Program Slicing and Sliding

Ran Ettinger
Wolfson College

Trinity Term, 2006

Submitted in partial fulfilment of the requirements for
the degree of Doctor of Philosophy

Oxford University Computing Laboratory
Programming Tools Group
Refactoring via Program Slicing and Sliding

Ran Ettinger, Wolfson College
Thesis submitted for the degree of Doctor of Philosophy
at the University of Oxford
Trinity Term, 2006

Abstract

Mark Weiser’s observation that “programmers use slices when debugging”, back in 1982, started a new field of research. Program slicing, the study of meaningful subprograms that capture a subset of an existing program’s behaviour, aims at providing programmers with tools to assist in a variety of software development and maintenance activities.

Two decades later, the work leading to this thesis was initiated with the observation that “programmers use slices when refactoring”. Hence, the thesis explores ways in which known refactoring techniques can be automated through slicing and related program analyses.

Common to all slicing related refactorings, as explored in this thesis, is the goal of improving reusability, comprehensibility and hence maintainability of existing code. A problem of slice extraction is posed and its potential contribution to refactoring research highlighted. Limitations of existing slice-extraction solutions include low applicability and high levels of code duplication.

Advanced techniques for the automation of slice extraction are proposed. The key to their success lies in a novel program representation introduced in the thesis. We think of a program as a collection of transparency slides, placed one on top of the other. On each such slide, a subset of the original statement, not necessarily a contiguous one, is printed. Thus, a subset of a statement’s slides can be extracted from the remaining slides in an operation of sideways movement, called sliding. Semantic properties of such sliding operations are extensively studied through known techniques of predicate calculus and program semantics.

This thesis makes four significant contributions to the slicing and refactoring fields of research. Firstly, it develops a theoretical framework for slicing-based behaviour-preserving transformations of existing code. Secondly, it provides a provably correct slicing algorithm. This application of our theory acts as evidence for its expressive power whilst enabling constructive descriptions of slicing-based transformations. Thirdly, it applies the above framework and slicing algorithm in solving the problem of slice extraction. The solution, a family of provably correct sliding transformations, provides high levels of accuracy and applicability. Finally, the thesis outlines the application of sliding to known refactorings, making them automatable for the first time.

These contributions provide strong evidence that, indeed, slicing and related analyses can assist in building automatic tools for refactoring.
To Dana, Amir, Zohara and Ze’ev; and to loved Bärbel.
I would first like to thank prof. Oege de Moor, for taking on the tough role of supervising me and my work. Not without struggle, we have finally come through, winning. In not hassling me while I was slowly progressing, and in acting as devil’s advocate, Oege has strongly contributed to the development and success of this work. I also thank Oege for introducing me to his fine group of students — some of whom will remain forever my friends (I hope) — and for introducing me to Dijkstra’s work on program semantics. I finally thank Oege for his effort in securing some funding for this work, after Intercomp’s unsurprising withdrawal, on my arrival to Oxford.

Mike Spivey supervised my work during Oege’s 2003 Sabbatical and influenced my direction tremendously. I’m especially grateful to Mike for the full attention and presence during our supervision meetings, always leaving me inspired and with fresh ideas. Jeremy Gibbons and Mike were my transfer examiners and later Jeremy and Jeff Sanders performed my confirmation of status. I am greatly indebted to all three for the insightful comments and feedback, which left a huge mark on the later results leading to this thesis.

The Programming Tools Group at Oxford, of which I was a member, provided a strong working environment, through weekly meetings, where talks could be practiced and ideas could be shared and discussed, and through joint work. I’m grateful to past and present members of the group, in particular to Iván Sanabria, Yorck Hünke, Stephen Drape, Mathieu Verbaere, and Damien Sereni, for the professional assistance and collaboration, for the mental support on rough days, and for the friendship. The collaborative highlight of my Oxford time was the work with Mathieu during his MSc project. I loved our endless discussions on programming and whatever, and I’m glad you and Dorothee finally returned for the DPhil, and became close friends. Thank you!

Other comlab (past) students, like Gordon, Fatima, Eran, Silvija, Jussi, Abigail, Penelope, Eldar, David, and Edouard, have contributed immensely to this professional and personal voyage. I’d also like to extend warm thanks to comlab’s administrative staff, for their continuously diligent support.

Big thanks go to my final examiners, Jeremy Gibbons and Mark Harman, for the interesting discussion during a good-spirited viva, and for their valuable suggestions, making this thesis a
better and more professional scientific report. Special thanks go to Raghavan Komondoor, Yvonne Greene, Iván Sanabria, Steve Drape, Sharona Meushar and Itamar Kahn, for commenting on final drafts of the thesis.

Intercomp Ltd. in Israel was where I first came up with the ideas for this research. I thank my colleagues and friends there, and I thank prof. Alan Mycroft of Cambridge for his contribution to the development of the initial research proposal. I also thank prof. Mooly Sagiv for introducing me to the academic world of program analysis, as well as Eran Tirit and Dr. Mayer Goldberg, for supporting my Oxford application with letters of recommendation. Itamar Kahn was inspirational in his own way, and the first to recommend Oxford to me. Sharon Attia was instrumental in the acceptance of an eventual Oxford offer, and started that adventure with me.

Student life in Oxford is brilliant, mainly due to the distribution of all students into colleges. Wolfson College provided a beautiful and peaceful environment, perfect for my living and studying. I would like to thank all my Wolfsonian friends, the staff, members of the boat club, and most importantly members of our football club. (After all, football was my number one reason for choosing England.) Participating in sports competitions with the many other colleges, and once a year with our sister college in Cambridge, Darwin, provided some of the best moments of my fantastic DPhil experience. On the Jewish side of things, I warmly thank Rabbi Eli and Freidy Brackman for providing a friendly, social and educational environment in their thriving Oxford Chabad society. And as for nutrition, nothing compares to Oxford’s late night Kebab vans! Thank you all for taking good care of me.

Financially surviving the five years of research, as a self-funded student, required some fund raising. I acknowledge the financial support of IBM’s Eclipse Innovation Grant, the ORS Scheme, and Oxford University’s Hardship Fund. Sun Labs hired me for a magnificent California internship (thanks to Michael Van De Vanter, James Gosling, Tom Ball and Tim Prinzing). Continuous teaching appointments by Oxford’s Computing Laboratory and the Software Engineering Programme were fun to perform and provided the much needed extra funds (thanks to Silvija Seres, Jeremy Gibbons, Steve McKeever, the administrative staff, and all students). In the final 8 months I was fully supported by my parents (Toda Ima Aba!) and my new position at the IBM Haifa Research Lab (thanks to Dr. Yishai Feldman and the SAM group) helps paying back Wolfson College’s Senior Tutor loan (many thanks to Dr. Martin Francis).

Shai, Uri, Itamar & Anna, Yacove, Koby & Adi, Sharona, Becky, Hezi, Keren, and Yo’av, all helped keeping morale high by visiting and maintaining overseas friendships. My sister Dana, brother Amir, and their families, my parents, Zohara and Ze’ev, and the rest of the family, including our UK-based relatives, most notably Yvonne Greene and her lovely family in Banbury where I found a home away from home, were all extremely supportive in ways more than one; and indeed very patient. I am forever grateful!
And last but not least, I am happy to thank Georgia Barbara Jettinger for her love and immeasurable contribution to the success of this journey. And it was actually Bärbel’s slip of a tongue that triggered the invention of this thesis’ sliding metaphor. I am deeply grateful to Oxford (and Edouard and Raya) for introducing us. Following you on your fieldwork to Paris, for my year of writing-up, turned out brilliant. Génial!

Rani Ettinger,
15 June 2007,
Tel Aviv, Israel

*Slip sliding away*

*Slip sliding away*

*You know the nearer your destination*

*The more you slip sliding away.*

*Simon & Garfunkel*
# Contents

## 1 Introduction

1.1 Refactoring enables iterative and incremental software development . . . . . . . . . 1  
1.2 The gap: refactoring tools are important but weak . . . . . . . . . . . . . . . . . . 2  
1.2.1 Motivating example: Fowler’s video store . . . . . . . . . . . . . . . . . . . . 2  
1.3 Programmers use slices when refactoring . . . . . . . . . . . . . . . . . . . . . . . . 7  
1.4 Automatic slice-extraction refactoring via sliding . . . . . . . . . . . . . . . . . . 8  
1.5 Overview: chapter by chapter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9  
1.6 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10  

## 2 Background and Related Work

2.1 Refactoring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12  
2.1.1 Informal reality . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12  
2.1.2 Underlying theory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13  
2.2 Slicing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15  
2.2.1 Slicing examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16  
2.2.2 On slicing and termination . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16  
2.2.3 Slicing criterion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17  
2.2.4 Syntax-preserving vs. amorphous and semantic slices . . . . . . . . . . . . . . 17  
2.2.5 Flow sensitivity: backward vs. forward slicing . . . . . . . . . . . . . . . . . . 18  
2.2.6 Slicing algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18  
2.2.7 SSA form . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19  
2.3 Slice-extraction refactoring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21  

## 3 Formal Semantics: Predicate Transformers

3.1 Set theory for program variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26  
3.1.1 Sets and lists of distinct variables . . . . . . . . . . . . . . . . . . . . . . . . . . . 26  
3.1.2 Disjoint sets and tuples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27  
3.1.3 Generating fresh variable names . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 Predicate calculus</td>
<td>27</td>
</tr>
<tr>
<td>3.2.1 The state-space metaphor</td>
<td>27</td>
</tr>
<tr>
<td>3.2.2 Structures, expressions and predicates</td>
<td>28</td>
</tr>
<tr>
<td>3.2.3 Square brackets: the 'everywhere' operator</td>
<td>29</td>
</tr>
<tr>
<td>3.2.4 Functions and equality</td>
<td>29</td>
</tr>
<tr>
<td>3.2.5 Global variables in expressions, predicates and programs</td>
<td>30</td>
</tr>
<tr>
<td>3.2.6 Substitutions</td>
<td>30</td>
</tr>
<tr>
<td>3.2.7 Proof format</td>
<td>31</td>
</tr>
<tr>
<td>3.2.8 From the calculus</td>
<td>32</td>
</tr>
<tr>
<td>3.3 Program semantics</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1 Predicate transformers</td>
<td>33</td>
</tr>
<tr>
<td>3.3.2 Different types of junctivity</td>
<td>33</td>
</tr>
<tr>
<td>3.3.3 A definition of deterministic program statements</td>
<td>35</td>
</tr>
<tr>
<td>3.4 Program refinement</td>
<td>36</td>
</tr>
<tr>
<td>4 A Theoretical Framework</td>
<td>37</td>
</tr>
<tr>
<td>4.1 Preliminaries</td>
<td>37</td>
</tr>
<tr>
<td>4.1.1 On slips and slides: an alternative to substatements</td>
<td>38</td>
</tr>
<tr>
<td>4.1.2 Why deterministic?</td>
<td>39</td>
</tr>
<tr>
<td>4.1.3 On deterministic program semantics</td>
<td>39</td>
</tr>
<tr>
<td>4.1.4 On refinement, termination and program equivalence</td>
<td>39</td>
</tr>
<tr>
<td>4.1.5 Semantic language requirements</td>
<td>41</td>
</tr>
<tr>
<td>4.1.6 Global variables in transformed predicates</td>
<td>42</td>
</tr>
<tr>
<td>4.2 The programming language</td>
<td>44</td>
</tr>
<tr>
<td>4.2.1 Expressions, variables and types</td>
<td>44</td>
</tr>
<tr>
<td>4.2.2 Core language</td>
<td>44</td>
</tr>
<tr>
<td>4.2.3 Extended language</td>
<td>46</td>
</tr>
<tr>
<td>4.3 Laws of program analysis and manipulation</td>
<td>49</td>
</tr>
<tr>
<td>4.3.1 Manipulating core statements</td>
<td>49</td>
</tr>
<tr>
<td>4.3.2 Assertion-based program analysis</td>
<td>50</td>
</tr>
<tr>
<td>4.3.3 Manipulating liveness information</td>
<td>51</td>
</tr>
<tr>
<td>4.4 Summary</td>
<td>52</td>
</tr>
<tr>
<td>5 Proof Method for Correct Slicing-Based Refactoring</td>
<td>53</td>
</tr>
<tr>
<td>5.1 Introducing slice-refinements and co-slice-refinements</td>
<td>53</td>
</tr>
<tr>
<td>5.2 Variable-wise proofs</td>
<td>54</td>
</tr>
<tr>
<td>5.2.1 Proving slice-refinements</td>
<td>54</td>
</tr>
</tbody>
</table>
9 A Slicing Algorithm

9.1 Flow-insensitive slicing

9.1.1 The algorithm

9.1.2 Example

9.2 Make it flow-sensitive using SSA-based slides

9.2.1 Formal derivation of flow-sensitive slicing

9.2.2 An SSA-based slice is de-SSA-able

9.2.3 The refined algorithm

9.2.4 Example

9.3 Slice extraction revisited

9.3.1 The transformation

9.3.2 Evaluation and discussion

9.4 Summary

10 Co-Slicing

10.1 Over-duplication: an example

10.2 Final-use substitution

10.2.1 Example

10.2.2 Deriving the transformation

10.3 Advanced sliding

10.3.1 Statement duplication with final-use substitution

10.3.2 Slicing after final-use substitution

10.3.3 Definition of co-slicing

10.3.4 The sliding transformation

10.4 Summary

11 Penless Sliding

11.1 Eliminating redundant backup variables

11.1.1 Motivation

11.1.2 Example

11.1.3 Dead-assignments-elimination and variable-merging

11.2 Compensation-free (or penless) co-slicing

11.3 Sliding with penless co-slices

11.4 Summary

12 Optimal Sliding

12.1 The minimal penless co-slice
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1.1</td>
<td>A polynomial-time algorithm</td>
<td>127</td>
</tr>
<tr>
<td>12.2</td>
<td>Slice inclusion</td>
<td>131</td>
</tr>
<tr>
<td>12.3</td>
<td>The optimal sliding transformation</td>
<td>134</td>
</tr>
<tr>
<td>12.4</td>
<td>Summary</td>
<td>135</td>
</tr>
<tr>
<td>13</td>
<td>Conclusion</td>
<td>137</td>
</tr>
<tr>
<td>13.1.1</td>
<td>Replace Temp with Query</td>
<td>138</td>
</tr>
<tr>
<td>13.1.2</td>
<td>More refactorings</td>
<td>138</td>
</tr>
<tr>
<td>13.2</td>
<td>Advanced issues and limitations</td>
<td>140</td>
</tr>
<tr>
<td>13.3.1</td>
<td>Formal program re-design</td>
<td>142</td>
</tr>
<tr>
<td>13.3.2</td>
<td>Further applications of sliding: beyond refactoring</td>
<td>143</td>
</tr>
<tr>
<td>A</td>
<td>Formal Language Definition</td>
<td>145</td>
</tr>
<tr>
<td>A.1</td>
<td>Core language</td>
<td>145</td>
</tr>
<tr>
<td>A.2</td>
<td>Extended language</td>
<td>154</td>
</tr>
<tr>
<td>B</td>
<td>Laws of Program Manipulation</td>
<td>158</td>
</tr>
<tr>
<td>B.1</td>
<td>Manipulating core statements</td>
<td>158</td>
</tr>
<tr>
<td>B.2</td>
<td>Assertion-based program analysis</td>
<td>163</td>
</tr>
<tr>
<td>B.2.1</td>
<td>Introduction of assertions</td>
<td>163</td>
</tr>
<tr>
<td>B.2.2</td>
<td>Propagation of assertions</td>
<td>165</td>
</tr>
<tr>
<td>B.2.3</td>
<td>Substitution</td>
<td>170</td>
</tr>
<tr>
<td>B.3</td>
<td>Live variables analysis</td>
<td>174</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Introduction and removal of liveness informtion</td>
<td>174</td>
</tr>
<tr>
<td>B.3.2</td>
<td>Propagation of liveness information</td>
<td>175</td>
</tr>
<tr>
<td>B.3.3</td>
<td>Dead assignments: introduction and elimination</td>
<td>177</td>
</tr>
<tr>
<td>C</td>
<td>Properties of Slides</td>
<td>180</td>
</tr>
<tr>
<td>C.1</td>
<td>Lemmata for proving independent slides yield slices</td>
<td>185</td>
</tr>
<tr>
<td>C.2</td>
<td>Slide independence and liveness</td>
<td>187</td>
</tr>
<tr>
<td>D</td>
<td>SSA</td>
<td>194</td>
</tr>
<tr>
<td>D.1</td>
<td>General derivation</td>
<td>194</td>
</tr>
<tr>
<td>D.2</td>
<td>Transform to SSA</td>
<td>201</td>
</tr>
<tr>
<td>D.3</td>
<td>Back from SSA</td>
<td>213</td>
</tr>
<tr>
<td>D.4</td>
<td>SSA is de-SSA-able</td>
<td>218</td>
</tr>
</tbody>
</table>
D.5 An SSA-based slice is de-SSA-able ......................................................... 221

E Final-Use Substitution .............................................................................. 223
  E.1 Formal derivation ............................................................................... 223
  E.2 Lemmata for proving statement dup. with final use ......................... 226
  E.3 Stepwise final-use substitution ......................................................... 229

F Summary of Laws .................................................................................... 230
  F.1 Manipulating core statements .......................................................... 230
  F.2 Assertion-based program analysis ..................................................... 231
    F.2.1 Introduction of assertions ............................................................ 231
    F.2.2 Propagation of assertions ............................................................ 231
    F.2.3 Substitution .................................................................................. 232
  F.3 Live variables analysis ....................................................................... 232
    F.3.1 Introduction and removal of liveness information ....................... 232
    F.3.2 Propagation of liveness information .......................................... 233
    F.3.3 Dead assignments: introduction and elimination ....................... 233

Bibliography ................................................................................................. 234
Chapter 1

Introduction

1.1 Refactoring enables iterative and incremental software development

Programming is a relatively young discipline. In its earlier days, the leading so-called Waterfall methodology for software development involved separate phases for design and for actual implementation. This was based on the assumption that a system can be fully specified up-front, ahead of its implementation. Any later change was considered software maintenance, and involved its own separate set of processes.

Modern methodologies, in contrast, inherently accommodate for change by admitting a more iterative and incremental software development process. That is, throughout its lifecycle, software is developed and released in iterations. Each such iteration is typically targeting an increment in functionality. Thus, an iteration may involve any and all aspects of development, including design and coding. Examples include the Rational Unified Process (RUP) [34], eXtreme Programming (XP) [7] and other so-called agile methodologies [65, 67].

Refactoring [48, 20] is the process of improving the design of existing software. This is achieved by performing source code transformations that preserve the original functionality. The ability to update the design and internal structure of programs through refactoring enables change during the lifecycle of any software system. Thus, refactoring is key to the success of software development.

The premise, when refactoring, is that the design should be clearly reflected by the code itself. Thus, clarity of code is imperative. Indeed, the goal of many refactoring transformations (e.g. for renaming variables) is to improve the readability of code.

Another theme in refactoring is the removal of duplication in code. (As a system evolves, duplication creeps in e.g. by the common ‘quick-and-dirty’ practice of ‘copy-and-paste’ of existing code.) Such redundancies can and should be removed. This removal is achieved by refactoring
transformations geared at enhancing reusability of code (e.g. by extracting common code into new methods with the so-called Extract Method refactoring).

The refactorings in this thesis will indeed target both improved comprehensibility and enhanced reusability, in supporting the development and maintenance of quality software systems.

Modern software development environments, e.g. MS Visual Studio [68] and Eclipse [66], include useful support for some refactoring techniques. However, the incompleteness and, at times, incorrectness of those tools calls for progress in the underlying theory.

In what follows, we illustrate the promise of refactoring and the power of its supporting tools, on the one hand, while identifying the gap to be filled by this thesis, on the other.

1.2 The gap: refactoring tools are important but weak

In code, a function that yields a value without causing any observable side effects is very valuable. “You can call this function as often as you like” [20, Page 279]. Such a call is also known as a query.

A refactoring technique called Replace Temp with Query was introduced by Fowler [20] to turn the use of a temporary variable holding the value of an expression into a query. The benefit is increased readability (in the refactored version) and reusability (of the extracted computation). This scenario is indeed supported in e.g. Eclipse, as a special case of the Extract Method tool.

A more complicated case of Replace Temp with Query occurs when the temp is not assigned the result of an expression, but rather the result of a computation spanning several lines of code. If those lines are consecutive in code (i.e. contiguous), they can be selected by the user and again the Extract Method tool may handle them successfully. Unfortunately, this will not always be the case; instead, when the code for computing a temporary result is tangled with code for other concerns, it is said to be non-contiguous.

1.2.1 Motivating example: Fowler’s video store

The following example is taken (with minor changes) from Martin Fowler’s refactoring book [20], where all refactorings are performed manually. [3] The example concerns a piece of software for running a video store, focusing on the implementation of one feature: composing and printing a customer rental record statement. The statement includes information on each of the rentals made by a customer, and summary information; a sample statement is shown in Figure 1.1.

In the original implementation, the preparation of the text to be printed and the computation of the summary information are tangled inside a single method (see Figure 1.2). In fact, Fowler

---

1 The example itself, as well as a variation on the accompanying discussion, has appeared in a paper titled: “Untangling: A Slice Extraction Refactoring” [17] by the author of this thesis, co-authored with Mathieu Verbaere.
CHAPTER 1. INTRODUCTION

Figure 1.1: A sample customer statement.

```java
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Iterator rentals = _rentals.iterator();
    String result = "Rental record for " +
                    getName() + "\n";
    while (rentals.hasNext()) {
        Rental each = (Rental) rentals.next();
        // show figures for this rental
        result += "\t" + each.getMovie().getTitle();
        result += "\t" + each.getCharge() + "\n";
        frequentRenterPoints += each.getFRP();
        totalAmount += each.getCharge();
    }
    // add footer lines
    result += "Amount owed is " +
             totalAmount + "\n";
    result += "You earned " +
              frequentRenterPoints +
              " frequent renter points";
    return result;
}
```

Figure 1.2: A tangled statement method.
Figure 1.3: The `statement` method after extracting the computations of the `total charge` and frequent renter points.

starts off with a much longer `statement` method containing all the logic for determining the amount to be charged and the number of frequent renter points earned per movie rental. These results depend on the type of the rented movie (regular, children’s or new release) and the number of days it was rented for.

Fowler then gradually improves the design by factoring out that rental-specific logic (into the `Rental` class, which is not shown here). The suggested refactoring steps are motivated by the introduction of a new requirement, namely the ability to print an HTML version of the statement. A quick-and-dirty approach would be to copy the body of the `statement` method, paste it into a new `htmlStatement` method and replace the text-based layout control strings with corresponding HTML tags. This would lead to duplication of the code for computing the temporary `totalAmount` and `frequentRenterPoints` variables.

For brevity, we join the refactoring session at the stage Fowler calls Removing Temps (Page 26). At this stage the computations of `totalAmount` and `frequentRenterPoints` are factored
CHAPTER 1. INTRODUCTION

36  private double getTotalCharge() {
37      double result = 0;
38      Iterator rentals = _rentals.iterator();
39      while (rentals.hasNext()) {
40          Rental each = (Rental) rentals.next();
41          result += each.getCharge();
42      }
43      return result;
44  }

Figure 1.4: The extracted total charge computation.

46  public int getTotalFRP() {
47      int result = 0;
48      Iterator rentals = _rentals.iterator();
49      while (rentals.hasNext()) {
50          Rental each = (Rental) rentals.next();
51          result += each.getFRP();
52      }
53      return result;
54  }

Figure 1.5: The extracted frequent renter points computation.
out (see figures 1.3, 1.5 for the result of those two steps). Fowler describes the path by which this was achieved as “not the simplest case of Replace Temp with Query, totalAmount was assigned to within the loop, so I have to copy the loop into the query method”. Indeed, to-date, no refactoring tool supports such cases.

Here is an outline of the mechanical steps that need to be performed by a programmer, in the absence of tool support, for extracting the total charge computation:

1. In the statement method of Figure 1.2, look for the temporary variable that is assigned the result of the total charge computation. This is totalAmount which is declared to be of type double in line 14. Its final value is added to the customer statement in line 29.

2. Create a new method, and name it after the intention of the computation: getTotalCharge. Declare it to return the type of the extracted variable: double. See line 36 in Figure 1.4.

3. Identify all the statements that contribute to the computation of totalAmount. In this case these are the statements in lines \{14, 16, 19, 20, 25, 26\}.

4. Copy the identified statements to the new method. See lines 37 to 42 in Figure 1.4.

5. Scan the extracted code for references to any variables that are parameters to the statement method. These should be parameters to getTotalCharge as well. In this case, the parameter list is empty.

6. Look to see which of the extracted statements are no longer needed in the statement method and delete those. In this case, the while loop is still relevant, and therefore the statements in lines \{16,19,20,26\} cannot be deleted; instead, they are duplicated. Lines \{14,25\} are needed only in the extracted code and are therefore deleted. In Figure 1.3 they are shown as blank lines, for clarity.

7. Rename the extracted variable, totalAmount, in the extracted method, getTotalCharge, to result, and add a return statement at the end of that method (see line 43 in Figure 1.4).

8. Replace the reference to the result of the extracted computation with a call to the target getTotalCharge method (line 29 in Figure 1.3).


A refactoring tool could reduce the above scenario to (a) selecting a temporary variable (whose computation is to be extracted), and (b) choosing a name for the extracted method. The tool, in turn, would either perform the transformation, or reject it if behaviour preservation cannot be guaranteed. For example, note that the correctness of the transformation above depended on the immutability of the traversed collection rentals (thus allowing untangling of the three traversals).
CHAPTER 1. INTRODUCTION

An attempt at providing such a tool, in early stages of the research leading to this thesis, suffered several drawbacks. Firstly, in order to guarantee behaviour preservation, the identified preconditions (e.g. no global variable defined in the extracted code) were clearly stronger than necessary. Secondly, the levels of code duplication were, again, higher than necessary. The duplication is due to extracted statements (identified in step 3 above) that are not deleted from the original (see step 6). As usual, code duplication could be considered harmful in itself, but perhaps more importantly, it indirectly affected applicability.

A successful reduction in duplication and weakening of preconditions, thus leading to a refined and more generally applicable tool, required a careful and rigorous study of the many intricacies in this refactoring. Results of that study are reported in this thesis.

The complete video-store scenario, particularly the breaking up and distribution of the initially monolithic statement method, motivates and justifies Fowler and Beck’s big refactoring to Convert Procedural Design to Objects [20, Chapter 12]: “You have code written in a procedural style. Turn the data records into objects, break up the behavior, and move the behavior to the objects”.

The steps of turning procedural design to objects mainly involve introducing new classes, extracting methods, moving variables and methods (to the new classes), inlining methods and renaming variables and methods. All those are either straightforward or already supported by modern refactoring tools. It is the extraction of non-contiguous code (as in Replace Temp with Query) for which automation is missing and required.

However hope is not lost, as some solutions to extraction of non-contiguous code have been proposed and investigated. (In fact, as will be shown later, those tackle a problem different from the above, but closely related.) Inspired by those, we shall dedicate this thesis to the development of a novel solution; one that will benefit from the advantages of each of those, whilst highlighting and overcoming respective limitations.

The extraction of non-contiguous code, especially when dealing with the automation of steps 3 and 6 of the mechanics in the example above, lead us to the following observation.

1.3 Programmers use slices when refactoring

To untangle the desired statements from their context, one can employ program slicing [61, 64]. A program slice singles out those statements that might have affected the value of a given variable at a given point in the program. A typical scenario is one in which the programmer selects a variable (or set of variables) and point of interest, e.g. totalAmount at line 29, in the example above (Figure 1.2); a slicing tool, in response, computes the (smallest possible) corresponding slice, e.g. the non-contiguous code of lines \{14,16,19,20,25,26\}. This slice can then be extracted...
into a new method, as was the case in steps 3 and 4 of that example. The idea of using slicing for refactoring has been suggested by Maruyama [42].

Program slicing was invented, by Mark Weiser, for “times when only a portion of a program’s behavior is of interest” [61], and with the observation that “programmers use slices when debugging” [62]. According to Weiser, slicing is a “method of program decomposition” that “is applied to programs after they are written, and is therefore useful in maintenance rather than design” [61].

This is no longer true. In modern software development, as was mentioned earlier, some design is normally done on each and every development iteration. Thus, since code of earlier iterations is already available when designing further features (or corrections to existing ones), slicing can be useful there too.

Therefore, the research leading to this thesis started with the observation that slicing can be useful in daily program development activities, even outside its initial domain of software maintenance. As a first step towards such usage, and since refactoring presents such an interesting blend of design, existing code and behaviour-preserving transformations, this research was initiated with the question: “How can program slicing and related analyses assist in building automatic tools for refactoring?”

1.4 Automatic slice-extraction refactoring via sliding

We shall propose automation of the Replace Temp with Query refactoring in latter stages of this thesis. The solution will be composed of a number of behaviour-preserving steps, in a manner slightly different from the earlier mechanics of manual transformation. In the first step, a selected slice will be extracted from its so-called complement (i.e. code for the remaining computation).

The problem of slice extraction can be formulated as follows:

Definition 1.1 (Slice extraction). Let $S$ be a program statement and $V$ be a set of variables; extract the computation of $V$ from $S$ (i.e. the slice of $S$ with respect to $V$) as a reusable program entity, and update the original $S$ to reuse the extracted slice.

A novel solution shall be developed in the course of this thesis, thus automating slice-extraction. The automation will be based on a correct (i.e. behaviour-preserving) slicing algorithm. This algorithm will itself be based on a special program representation, specifically designed for capturing non-contiguous code. This representation’s primitive elements will be called slides. (This decomposition of a program into slides is in accordance with a program execution metaphor of overhead projection of programs printed on transparency slides; see Chapter 8.)

The author would like to gratefully acknowledge Prof. Alan Mycroft’s advice during preparation of the research proposal, particularly in the formulation of this research question.
It is in illustrating and formalising the slice-extraction refactoring that the program medium of slides will be instrumental. Suppose the code of a program statement $S$ is printed on a single transparency slide. Our initial solution begins by duplicating that slide, yielding two clones, say $S_1$ and $S_2$, and placing them one on top of the other. This is then followed by sliding one of the slides (say of $S_2$) sideways, and by adding so-called compensatory code. This compensation will be responsible for preserving behaviour.

Behaviour can be preserved by keeping initial values of all relevant variables (in fresh backup variables) ahead of $S_1$, and retrieving those after $S_1$ but ahead of $S_2$. Furthermore, extracted results, $V$, can be saved after $S_1$ and retrieved on exit from $S_2$. Pictorially, sliding of $S, V$ will turn $S$ into something like the following (with “ ; ” for sequential composition; note that the left column is composed with the right, thus for chronological order read the former, top-down, before moving on to the latter):

<table>
<thead>
<tr>
<th>(keep backup of relevant initial values)</th>
<th>(retrieve backup of initial values)</th>
</tr>
</thead>
<tbody>
<tr>
<td>; $S_1$ (first clone, i.e. extracted code)</td>
<td>; $S_2$ (second clone, i.e. complement)</td>
</tr>
<tr>
<td>; (keep backup of final values of the extracted $V$)</td>
<td>; (retrieve backup of final values)</td>
</tr>
</tbody>
</table>

A naive sliding transformation, in the form of full statement duplication (as described above), is formally developed in Chapter 6.

A number of improved versions of sliding, with the goal of reducing code duplication, will be explored and formalised throughout the thesis. Those will benefit from our decomposition of a program statement into smaller entities of non-contiguous code (i.e. slides, to be formalised in Chapter 8).

The reduction of duplication will be achieved by making both the extracted code (i.e. $S_1$ above) and the complement (i.e. $S_2$) smaller. In later improvements, the compensatory code will be made smaller too (see Chapter 11).

1.5 Overview: chapter by chapter

This opening chapter has introduced the challenge of slice-extraction untangling transformations, with the goals of improving readability and reusability of existing code. The importance and potential implications of this refactoring and its automation have been highlighted and briefly demonstrated through a known example from the refactoring literature. Finally, hints to our path for automating slice extraction have been given. The rest of this thesis is structured as follows:

- In Chapter 2 we present background material and related work. This includes refactoring,
slicing, and the application of slicing to refactoring in extraction of non-contiguous code.

- In Chapter 3 we give background to the adopted formal approach, introducing some relevant concepts from predicate calculus and predicate transformers, set theory and program refinement.

- In Chapters 4 and 5 we begin the presentation of original work by developing a formal framework for correct slicing-based refactoring, including the definition of a programming language, a collection of laws to facilitate program analysis and manipulation, and a method for proving general algorithmic refinements through newly introduced slicing-related ones.

- In Chapter 6 we take the first step towards slice extraction by formally developing a transformation of statement duplication. The result is a naive sliding transformation, with both the extracted code and complement being clones of the original statement.

- In Chapters 7, 8 and 9 we develop a first improvement of sliding. The semantic and syntactic requirements of slicing are derived, leading to the formalisation of a novel slicing algorithm, one that is based on a program representation of slides. With this slicer, both the extracted code and the complement are specialised to be the slice of extracted variables and the slice of the remaining defined variables, respectively.

- In Chapter 10 we target further reductions in the duplication caused by sliding. Those are based on the observation that the complement (or co-slice), previously being the slice of all non-extracted variables, can become smaller by reusing values of extracted variables.

- In Chapter 11 we target the identification and elimination of redundant compensatory code, result of earlier formulations of sliding.

- In Chapter 12 we pose and solve a couple of optimisation problems, thus yielding an optimal slice-extraction solution via sliding.

- Finally, we conclude in Chapter 13 by considering the application of sliding for automating known refactorings, discussing advanced issues and limitations, and suggesting possible directions for future work.

1.6 Contributions

This thesis brings together the fields of program slicing and refactoring. As such, it makes four significant contributions to those fields, as listed below.
1. It develops a theoretical framework for slicing-based behaviour-preserving transformations of existing code. The framework, based on wp-calculus, includes a new proof method, specifically designed to support slicing-related transformations of deterministic programs. The framework further includes a novel program decomposition technique of program slides, aiming to capture non-contiguous code.

2. It provides a provably correct slicing algorithm. This application of our theory acts as evidence for its expressive power whilst enabling constructive descriptions of slicing-based transformations.

3. It applies the above framework and slicing algorithm in solving the problem of slice extraction. The solution takes the form of a family of provably correct sliding transformations. Drawing inspiration from a number of existing solutions to related problems of method extraction, sliding is successful in providing high levels of accuracy and applicability.

4. It identifies and outlines the application of sliding to known refactorings, making them automatable for the first time. Examples of such refactorings include Replace Temp with Query and Split Loop.

These contributions provide strong evidence for the validity of our research question. Indeed, slicing and related analyses can assist in building automatic tools for refactoring.
Chapter 2

Background and Related Work

2.1 Refactoring

2.1.1 Informal reality

Refactoring is defined informally as the process of improving the design of existing software systems. The improvement takes the form of source code transformations. Each such transformation is expected to preserve the behaviour of the system while making it more amenable for change. A programmer can refactor either manually or with the assistance of automatic tools.

Refactoring was introduced by William Opdyke in his PhD thesis [48] and later became widely known with the introduction of Martin Fowler’s book [20].

The refactoring.com website [71] maintains a list of refactoring tools and an online catalog of refactorings [69]. The refactoring community discusses the techniques, tools and philosophy on the refactoring mailing list [72].

In [69, 20], around 100 refactoring techniques are described. There are simple refactorings such as renaming a class and some more complicated ones, e.g. for extracting a class or a method, or for moving a method from one class to another. Some bigger refactorings may involve a whole hierarchy of classes, for example introducing polymorphism, collapsing a redundant class hierarchy, or even as complex and ambitious as converting a program with procedural design to a more object-oriented one.

Being driven mostly by examples, the description of each refactoring, in [69, 20], is fairly informal and imprecise. The success of each transformation depends on the programmer’s good judgement, complemented with expected assistance from the compiler and the availability of a comprehensive suite of automated tests.

Eliminating that unconvincing dependence on testing is one of the challenges of refactoring tools. Such a tool is typically interactive; the programmer is responsible to select a specific refac-
toring from the menus, the tool in response performs the transformation, asking the programmer to fill in any required details such as new names for introduced program elements.

Another (related) goal of refactoring tools is to speed up the process of refactoring, thus supporting improved productivity of programmers. Ultimately, programmers would trust the tools, employ them frequently, on a daily basis, as is dictated by requests for change in the existing software on which they work.

The RefactoringBrowser for Smalltalk, developed by John Brant and Don Roberts at the University of Illinois [53], was the first designated refactoring tool. Its success was followed by several attempts to develop refactoring tools for the Java programming language [25], including IntelliJ’s IDEA, Microsoft’s Visual Studio and (initially IBM’s) Eclipse. Those tools support some of the offered refactorings, such as Move/Pull-Up/Push-Down/Extract/Inline Method, Rename Field/Method/Class, Self-Encapsulate Field, Add Parameter, and Extract Interface.

However, that support is far from perfect, as short experiments we performed (first in 2003 and then again in 2005 [70, 55]) revealed. There, it was demonstrated how modern tools are particularly weak in supporting cases where non-trivial data-flow and control-flow analyses are required. These shortcomings led, in some cases, to an apparently successful refactoring that was yielding grammatically incorrect code; in other cases, potentially correct transformations were unnecessarily rejected due to inaccurate, and at times incorrect analysis.

Such bugs in refactoring tools call for a review of refactoring theory. Their existence also act as motivation for the formal approach taken in this thesis.

2.1.2 Underlying theory

Program representation and analysis

As a result of developing several versions of the Smalltalk RefactoringBrowser, Roberts [52] identified several criteria, both technical and practical, necessary to the success of a refactoring tool. The technical requirement is that the tool must maintain a program database, that holds all the required information about the refactored program’s entities, e.g. packages, classes, fields, methods and statements, and also their relations and cross-references. The database should enable the tool to check properties of the program both when checking whether a refactoring request is legal, and in performing the transformation. As the source code may constantly change, either manually by the programmer or by the refactoring (or any other source code manipulation) tool, the program database must also be constantly updated. Regarding the techniques that can be used to construct the program database, Roberts states that “at one end of the scale are fast, lexical tools such as grep. At the other end are sophisticated analysis techniques such as dependency graphs. Somewhere in the middle is syntactic analysis using abstract syntax trees. There are tradeoffs between
speed, accuracy, and richness of information that must be made when deciding which technique to use. For instance, grep is extremely fast but can be fooled by things like commented-out code. Dependency information is useful, but often takes a considerable amount of time to compute.”

Existing tools mostly use the abstract syntax tree (AST) compromise, whereas the analysis required for transformations in this thesis will be of the kind applied in constructing dependency graphs. In doing so, and in light of Roberts’ observation, as stated above, we pay some attention to efficiency and performance, when constructively expressing algorithms for analysis and transformation. In particular, most of those will indeed be tree based and require only one pass over an analysed program’s AST. (This is made possible by the simplicity of our supported language.)

However, behaviour preservation, as is discussed next, will be our prime goal. Consequently, we shall be concerned with correctness of our algorithms more than with their corresponding performance and complexity.

**Behaviour preservation**

Roberts further discusses the accuracy property expected from a refactoring tool. He argues that the refactorings that a tool implements must reasonably preserve the behaviour of programs, as total behaviour preservation is impossible to achieve. “For example, a refactoring might make a program a few milliseconds faster or slower. Usually, this would not affect a program, but if the program requirements include hard real-time constraints, this could cause a program to be incorrect”. The reasonable behaviour-preservation degree that should be expected from a refactoring tool was formally defined by Opdyke [48] as a list of seven properties two versions of a program must hold before and after a refactoring. The first six involve syntactical correctness properties that are necessary for a clean compilation of both versions of the program. The seventh property is called “Semantically equivalent references and operations”, and is defined as follows: “Let the external interface to the program be via the function main. If the function main is called twice (once before and once after the refactoring) with the same set of inputs, the resulting set of output values must be the same” [48].

This property, when dealing with terminating sequential programs, corresponds to the concept of refinement (see Section 3.4 in the next chapter). And indeed, in his PhD thesis (“Refactorings as Formal Refinements” [11]), Márcio Cornélio formalised a large number of Fowler’s refactorings as “algebraic refinement rules involving program terms”. The supported language (ROOL, for “Refinement Object-Oriented Language”) is said to be “a Java-like object-oriented language” with formal semantics based on weakest preconditions (see Chapter 3 ahead).

However, Cornélio does not support the refactoring for removing temps (Replace Temp with Query) which is targeted by this thesis. To formalise and solve such refactoring problems, the original refinement calculus approach, as presented by Morgan [15] and others, needs to be com-
bined with projection onto a subset of the program variables, as we discuss in Chapter 5. Thus, like Cornélio, we base this work on formal refinement and weakest-preconditions semantics.

For simplicity, and due to the intricate nature of the problem, we shall target a very simple imperative language, rather than a fully object-oriented one. For the same reasons, we shall focus on preservation of semantics alone, while avoiding all (important for themselves) questions over syntactic validity of transformed programs (as e.g. expressed in Opdyke’s first six properties).

### Composition of refactorings

Opdyke defined high-level refactoring techniques as a composition of lower-level ones. Each low-level refactoring is defined with a corresponding set of preconditions. Those, expressed in first order logic over predicates available in the program database, must be satisfied by the program and the refactoring criterion (i.e. the type of refactoring and the accompanying parameters chosen by the user) before a correct refactoring can be performed. The refactoring tool is responsible for performing such checks.

A naive implementation of refactoring composition would update the program database after every step. When the composition consists of a long sequence of refactorings, this may yield an inefficient and slow tool. One approach for reducing the amount of analysis in the implementation of composite refactorings was introduced in [52]. There, each refactoring’s definition was augmented with a set of properties that a program will definitely satisfy after the transformation, i.e. postconditions. Using this information, after each step of the composed refactoring, the program database can be incrementally updated, rather than be re-computed from scratch.

The approach taken in this thesis, however, is somewhat different. Indeed, our transformations shall be composed of (at times exceedingly long) sequences of smaller steps. But instead of expecting the actual tool to perform each and every step, we shall first formally develop a solution “by hand”; then overall preconditions shall be carefully collected; thus the bigger refactorings shall be formally derived from existing, smaller ones, hence potentially leading to more efficient tools.

As was mentioned in the preceding chapter, refactoring tools can benefit from the capabilities of a decomposition technique known as program slicing. For this we now turn to present relevant background on slicing, before relating the two (in the section to follow).

### 2.2 Slicing

Program slicing is the study of meaningful subprograms. Typically applied to the code of an existing program, a slicing algorithm is responsible for producing a program (or subprogram) that preserves a subset of the original program’s behaviour. A specification of that subset is known as a slicing criterion, and the resulting subprogram is a slice.
Slicing was invented with the observation that “programmers use slices when debugging” [62]. Nevertheless, the application of program slicing does not stop there. Further applications include testing [29, 8], program comprehension [9, 51], model checking [44, 15], parallelisation [63, 5], software metrics [50, 43], as well as software restructuring and refactoring [40, 42, 17]. The latter application is considered in this thesis.

There can be many different slices for a given program and slicing criterion. Indeed, there is always at least one slice for a given slicing criterion: the program itself [61]. However, slicing algorithms are usually expected to produce the smallest possible slices, as those are most useful in the majority of applications.

### 2.2.1 Slicing examples

Here is a variation on the “hello world” of program slicing, computing and printing the sum and product of all numbers in a given array of integers. The index of each statement is given to its left, for later reference.

```
<table>
<thead>
<tr>
<th>original</th>
<th>slice of sum from 8</th>
<th>slice of prod from 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i := 0</td>
<td>i := 0</td>
</tr>
<tr>
<td>2</td>
<td>; sum := 0</td>
<td>; sum := 0</td>
</tr>
<tr>
<td>3</td>
<td>; prod := 1</td>
<td>; prod := 1</td>
</tr>
<tr>
<td>4</td>
<td>; while i&lt;a.length do</td>
<td>; while i&lt;a.length do</td>
</tr>
<tr>
<td>5</td>
<td>sum := sum+a[i]</td>
<td>sum := sum+a[i]</td>
</tr>
<tr>
<td>6</td>
<td>; prod := prod*a[i]</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>; i := i+1</td>
<td>; i := i+1</td>
</tr>
<tr>
<td></td>
<td>od</td>
<td>od</td>
</tr>
<tr>
<td>8</td>
<td>; out &lt;&lt; sum</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>; out &lt;&lt; prod</td>
<td></td>
</tr>
</tbody>
</table>
```

A slice of sum from statement 8 must contain the statements \{1, 2, 4, 5, 7\}, and thus can be obtained by deleting the irrelevant statements \{3, 6, 8, 9\}. Similarly, a backward slice of prod from 9 should contain \{1, 3, 4, 6, 7\}, and can be obtained by deleting \{2, 5, 8, 9\}.

### 2.2.2 On slicing and termination

An interesting aspect of program behaviour is that of termination. Do we expect a slice to preserve conditions for termination? For example, should the loop statement in the program above be included in the slice for the array `a` from statement 9? And what if `a.length` is negative?
When conditions for termination are preserved, the slice is said to be termination sensitive. Such slices have been applied e.g. in model reduction. However, in an attempt to yield the smallest possible slices, it is common to remove non-affecting code even if this code might not terminate. This way, the empty statement `skip` is a valid slice for the array `a` in the example above. Thus, slicing may introduce termination, as is incidentally the case with refinement (see Section 3.4 of the following chapter).

2.2.3 Slicing criterion

An interactive tool for slicing can be seen as an extension to a source code editor, where the user can select where to slice from and the tool answers by highlighting the set of statements in the slice. The user selection, i.e. the slicing criterion, can be specified in different ways. In Weiser’s original definition, it was a pair, \( \langle i, V \rangle \), combining a program point, \( i \), and a set of variables, \( V \), e.g. \( \langle 8, \{ \text{sum} \} \rangle \) in the example above. A simplified version, \( \langle i \rangle \), is obtained by omitting the set of variables, e.g. \( \langle 8 \rangle \). There, all the variables that are used in the selected substatement are of interest (e.g. \( \langle 8, \{ \text{out, sum} \} \rangle \) above). Some slicing algorithms (most notably the PDG-based slicers), support such kind of criteria exclusively.

A third variation of the slicing criterion formulation is obtained by selecting a (possibly compound) statement and a set of variables of interest. Here, by avoiding any mention of a program point, we mean to slice from the end of the selected statement. For example, the slice of \( \langle 8, \{ \text{sum} \} \rangle \) (on the second column above) is a valid slice of \( \langle S, \{ \text{sum} \} \rangle \) (where \( S \) stands for the compound statement 1-9), whereas the slice for \( \langle T, \{ \text{sum} \} \rangle \) (where \( T \) is the compound statement of 1-3) would consist of substatement 2 alone. Similarly, the slice for \( \langle S, \{ \text{out} \} \rangle \) is the full \( S \). (Note that here the scope for slicing is mentioned explicitly whereas otherwise it is implicitly expected to be the whole program.) This kind of criteria has appeared e.g. in [59] and is used, exclusively, in this thesis.

2.2.4 Syntax-preserving vs. amorphous and semantic slices

When the slice is limited to constructs from the original program, it is said to be syntax preserving. Such slices are constructed by deleting irrelevant statements from the original program. Thus, a syntax-preserving slice of a given program statement corresponds to a substatement, possibly a non-contiguous one.

Amorphous slicing, in turn, combines slicing with a range of transformations, in simplifying
the resulting code (see e.g. [27]). For example, a termination-sensitive slice of the array \( a \), above, would potentially be able to exclude the loop from the slice, replacing it with a single test of the length of \( a \). This way, the initialisation of variable \( i \) would be successfully and correctly removed.

Semantic slicing (defined e.g. in [59]), defines the semantic requirements as expected from slices, and accepts any program that meets those requirements as a valid slice. When constructively computing semantic slices, similar techniques to those of amorphous slicing are used, in simplifying the result.

### 2.2.5 Flow sensitivity: backward vs. forward slicing

When a program analysis result depends on the order of the statements (i.e. when the analysis of a program \( " S1 ; S2 \) " is expected to differ from the analysis of \( " S2 ; S1 \) "), the analysis is said to be flow-sensitive [47]. For producing smaller, more accurate slices, a slicing algorithm should be flow-sensitive. As such, it can be applied in one of two directions.

Traditional slicing, as presented so far, is known in that respect as backward slicing. Indeed, as is the case with backward analysis [47, 54], its algorithms propagate information against the flow of control, while answering questions of what might have happened before arriving at a program point. The complementary technique is called forward slicing and is computed by looking forward from a selected program point, thus answering the question “which statements may be affected by the value computed at this point?”

A slicing algorithm to be developed in this thesis, in Chapter 9, will compute backward slices. An initial version will be flow-insensitive. Then, flow-sensitivity will be gained by transforming a program to and from a static-single-assignment (SSA) form, before and after the slicing, respectively. Background on the SSA form will be given shortly, but first we turn to discuss slicing algorithms.

### 2.2.6 Slicing algorithms

In this thesis we shall target intraprocedural, backward, syntax-preserving and static slicing. A variety of such algorithms exists. Those are based on a wide range of program representations, from abstract syntax trees (AST) [19, 18], through value dependence graphs (VDG) [16, 60] and the static single information (SSI) form [54], to control flow graphs (CFG) [61, 64] and the program dependence graph (PDG) [49, 33, 6]. We briefly mention Weiser’s original approach and the PDG-based one.

According to Weiser [61], automatic slicing should be performed on the program’s flow graph [1] using data-flow equations. Those approximate the set of variables that may be (directly or indirectly) relevant for a given slicing criterion at each node of the graph. A node \( n \) is added to the
slice if it defines (i.e. may modify the value of) any of those relevant variables (associated with \( n \)). Furthermore, any “branch statement which can choose to execute or not execute” another node which is already in the slice, should, in general, be added to the slice [61]. Thus, both data-flow and control-flow influences are taken into account, iteratively, until a fixed point is reached.

When the slicing criterion involves all variables that are referenced in a selected program point, the program dependence graph (PDG) offers a fast algorithm for computing the corresponding slice. The PDG, like the flow graph, contains a node for each program statement; the directed edges (relevant for slicing) correspond to direct data-flow and control-flow influences, respectively. Thus, slicing is reduced to a reachability problem. Each node from which there is a directed path to the selected node (that correspond to the slicing criterion), is added to the slice. This solution is particularly effective in a situation in which many slices are to be computed on the same program, since the time to construct the PDG can hence be amortised over all slice computations.

Note that both Weiser’s and the PDG-based approaches consider control and data dependences on the same level. That is, at each step of the respective algorithm, both kinds of influences are taken into account in adding statements to the computed slice. That choice is challenged in this thesis.

As an alternative, we shall primarily consider control dependences, in producing program entities called slides (see Chapter 8). Those slides shall than be treated as primitives in a novel slicing algorithm (Chapter 9) that involve data dependences (or rather slide dependences) only. Thus, when interested in slices from the end of a program, our algorithm will yield traditional slices, as in the algorithms above, on the one hand, while producing potentially smaller statements, for what we call co-slices (Chapter 10), on the other.

Our program representation of slides and hence the slicing algorithm will benefit from the popular intermediate representation (IR) of static single assignment (SSA) [12, 54], which is introduced next.

2.2.7 SSA form

Static single assignment form is an intermediate program representation in which “every variable assigned a value in it occurs as the target of only one assignment” [46]. As such, it has proved useful in static program analysis, in particular for implementing fast and effective optimizing compilers [46]. Typically, a program is translated into SSA form before performing some program analyses and related transformations (e.g. constant propagation, invariant code motion); once done, the transformed program is translated back to its original form.

Jeremy Singer defines the SSA form, along with its younger sister of static single information (SSI), as members of a more general family of IRs, of virtual register renaming schemes (VRRS) [54]. Any VRRS family member can be generally described as a control flow graph (CFG) with a
When a defined register (or variable) is renamed, all its references must be renamed too. Thus, each reference must be reached by a single corresponding definition. This is achieved by adding so-called pseudo variables in merge points of the CFG. Those merge all reaching definitions into one new name, using a so-called $\phi$-function. For example, on exit from an IF statement, $x_3 := \phi(x_1, x_2)$ would merge the two values $x_1$ and $x_2$ (we call them instances and accordingly $x_3$ a pseudo instance) of the two branches of the IF. The merge is such, that the $x_1$ will be taken on arrival from one branch, and the $x_2$ if arriving from the other. In general, there are as many arguments to a $\phi$-function as there are incoming edges in the control flow.

In our context of source-to-source transformations, supporting a simple language with structured control flow and no aliasing (as will be explained later), we shall be interested in an SSA-like renaming on the level of program variables (rather than virtual registers). For formalising and giving examples of SSA, and since our $\phi$-functions will always require two arguments, we shall avoid them altogether. Instead, they will be pushed back to the incoming branches, separated into two individual assignments (e.g. $x_3 := x_1$ at the end of one branch of an IF statement, and $x_3 := x_2$ at the other). Accordingly, a variable that is assigned (and used) in a DO loop, will have a designated pseudo instance assigned both before the loop and at the end of its body. For example, the SSA version of the \textit{sum} and \textit{prod} program from above will be as follows:

\begin{verbatim}
  i_1 := 0
  ; sum_2 := 0
  ; prod_3 := 1
  ; i_4,sum_4,prod_4 := i_1,sum_2,prod_3
  ; while i_4<a.length do
    sum_5 := sum_4+a[i_4]
    ; prod_6 := prod_4*a[i_4]
    ; i_7 := i_4+1
    ; i_4,sum_4,prod_4 := i_7,sum_5,prod_6
  od
  ; out_8 := out ++ sum_4
  ; out_9 := out_8 ++ prod_4
\end{verbatim}

Note that appending to the \textit{out} stream had to be simplified (with ++ taking the place of $\ll$). Note also that, following SSA tradition, we rename instances by adding a subscript. However, we deviate from the common practice of using increasing natural numbers for the instances of each variable. Instead, in program examples we prefer to use, for each assignment, its (informal) label
as a unique subscript (for all variables defined in it). Thus, our loop pseudo instances (e.g. $i_4$) are defined in two distinct places, on the one hand, but both define the same instance, on the other.

The SSA form will be formalised in Section 8.6 and Appendix D and applied for slicing and sliding. In particular, our new slicing algorithm is based on slides of the SSA form. This slicing algorithm should in turn be useful in automating slice-extraction transformations. Accordingly, we now turn to present background material on that problem.

### 2.3 Slice-extraction refactoring

An interactive process for behaviour-preserving method extraction was first proposed by Opdyke [48] and (independently) by Griswold and Notkin [26]. This was however restricted to the extraction of contiguous code.

Maruyama, in a paper titled “Automated Method-Extraction Refactoring by Using Block-Based Slicing” from 2001 [42], proposed a scenario according to which, extraction is performed on a block of statements (i.e. a compound statement — we shall simply call it a statement — acting as scope for extraction) and a user selected variable whose computation is to be extracted from the remaining computation. A slice of the selected variable in the selected scope is extracted into a new method; a call to that method is placed ahead of the code for the remaining computation. We call the latter the *complement*, borrowing Gallagher’s terminology [22]. Maruyama’s rudimentary solution is described formally; the importance of proving correctness is highlighted, but no proof is given.

Earlier research on extracting slices from existing systems, in the context of software reverse engineering and reengineering, including that of Lambile and Visaggio [41] and Cimitile et al. [10], has focused mainly on how to discover reasonable slicing criteria. In our context of refactoring, we prefer to leave the choice of what to extract to the programmer. Our approach, in turn, will focus mainly on how to reorganise the original code to benefit from the extraction.

In what follows, we explore state-of-the-art solutions to a related problem, we call it arbitrary method extraction, according to which, instead of extracting the slice of a variable, a set of (not necessarily contiguous) statements is selected for extraction.

Lakhotia and Deprez defined an arbitrary-method-extraction transformation called Tuck [40]. In tucking, a selection of statements and scope for extraction is made (either by the user or some other tool), and a tool, in response, computes the slice (of the selected statements) and complement, and composes them sequentially, along with some compensation that may include backup of initial values and variable renaming (in the complement). The complement itself is computed through slicing from all non-extracted statements (in the selected scope). If both the slice and its complement define a variable that is live on exit from scope, the transformation is
rejected.

Suppose we are asked to extract the slice of statement 2, in the following:

```c
1; while i<a.length do
2   sum := sum+a[i]
3   ; prod := prod*a[i]
4   ; i := i+1
od
```

Tucking would compute a statement made of \{1, 2, 4\} as the slice to be extracted; from the remaining statement, 3, it would compute \{1, 3, 4\} as basis for the complement; thus so long as the variable \(i\) is not live-on-exit (i.e. will not be used before being re-defined after the loop), we would get something like:

```c
ii := i

; while ii<a.length do
   sum := sum+a[ii]
   ; ii := ii+1
od

; while ii<a.length do
   prod := prod*a[ii]
   ; ii := ii+1
od
```

The final step of Tuck would then fold the extracted slice into a reusable method.

The 2000 version of arbitrary method extraction by Komondoor and Horwitz (to be referred to as KH00, [38]), is particularly effective in reordering statements. A sequence of statements is selected as scope, and a subset of those is selected for extraction. The algorithm seeks valid permutations of the sequence in which the selected statements are grouped together (i.e. forming contiguous code). The validity of a permutation depends on some control flow and data flow related constraints.

For example, suppose we are asked to extract the computation and printing of \(sum\), on statements \{2, 3, 7\} in the following:
CHAPTER 2. BACKGROUND AND RELATED WORK

The KH00 would successfully yield the following two alternatives:

In comparison to tucking, this algorithm extracts precisely the selected statements, even if those do not form a complete slice. This is made possible by allowing two complementary parts: one to be executed before the extracted code, the other after.

According to this algorithm, neither duplication nor compensatory code is permitted. Consequently, in cases where no permutation satisfies all constraints, the transformation is rejected.

In their 2003 version of arbitrary method extraction [39], Komondoor and Horwitz have relaxed their earlier restriction on duplication: this time predicates (as well as jumps, which are outside the scope of this thesis) are allowed to be duplicated, while other statements, e.g. assignments, are not. Furthermore, this time, instead of having to reject some transformation requests, when no permutation satisfies all ordering constraints, the new strategy is to drag problematic statements along with the selected statements, to the extracted part of the resulting program. They refer to
that dragging as promotion.

A first criticism of such promotion, by Harman et al., has appeared in [28], where amorphous slicing has been suggested for procedure and function extraction. Their target was to support program comprehension. Accordingly, their transformations are exploratory, with no aim to keep the resulting program, as we do in refactoring.

Another technique of code untangling for program comprehension, called fission (reverse of fusion), has been suggested by Jeremy Gibbons [24]. With fission, the design of a program can be reconstructed from its implementation. Gibbons illustrate the approach using code examples from the slicing literature [22]. Indeed, according to Gibbons, “slicing is a fission transformation, reversing the fusion of independent but similarly-structured computations”. In contrast to program comprehension, in refactoring the focus is on automation with syntax preservation, such that the resulting program would reflect an update in the design whilst being easily recognised by the programmer.

As with Harman et al. and Gibbons, the KH03 is not (and was not designed to be) a slice-extraction algorithm. (It was actually designed for reducing code duplication by eliminating multiple clones of code, replacing them all with method calls.) For example, KH03 cannot untangle the computation of \( \text{sum} \) from \( \text{prod} \), as the Tuck transformation does, since that would involve a duplication of the assignment to \( i \). In general, in KH03, any loop would have to be either completely extracted or not at all.

In comparison to its predecessor KH00, the KH03 algorithm presents a step forward in the sense that some duplication is allowed, thus rendering it more applicable (in fact it is totally applicable, as in the worst case it extracts the whole program in scope). It offers another improvement with respect to jumps (which are, again, outside the scope of our investigation). However, in an attempt to make it more scalable (its complexity is polynomial, compared with the exponential KH00), it might extract more statements. In the example of KH00 above, the KH03 algorithm would fail to move the computation of \( \text{prod} \) out of the way; instead it would extract it along with the selected statements.

Nevertheless, the KH03 algorithm offers one improvement over its predecessors, which is relevant for slice extraction. Komondoor and Horwitz criticise (in [39]) the Tuck transformation for not allowing data to flow from the extracted slice to its complement. This results in too large complements, as is demonstrated in the next example. Suppose we are asked to extract statements \( \{1, 2, 4, 6\} \) (i.e. the slice of \( \text{out} \)) from the following program (on the left). The KH03 would yield, in response, the version on the right:
Note that tucking, on this example, would have duplicated the entire computation of \textit{sum}, whereas the KH00 algorithm would have failed, since the selection does not form a valid sequence.

The challenge of this thesis will be to combine the untangling abilities of Tuck with improved applicability and reduced levels of code duplication, as in KH03, thus yielding (for the example above, and even in a case where \textit{i} is live-on-exit) something like:

This completes our presentation of background material and related work on the topics of refactoring, slicing and slicing-based refactoring. Our approach to solving the problem of slice extraction is based on formal semantics using so-called predicate transformers. The next chapter, our second and last background chapter, will introduce relevant concepts and basic theory.
Chapter 3

Formal Semantics: Predicate Transformers

This chapter introduces background material on the formal approach for program semantics to be adopted by this thesis. It is mainly based on Dijkstra and Scholten’s monograph *Predicate Calculus and Program Semantics* [13] (to be referred to as DS). Relevant properties and theorems will be recalled. Those will later be used in formally developing our framework of correct transformations. Furthermore, background on the concept of refinement and its relevance for refactoring and slicing is presented.

3.1 Set theory for program variables

Some operations and properties from set theory will be useful in discussing sets of program variables and in calculating program properties.

3.1.1 Sets and lists of distinct variables

For simplicity and convenience, we will interchangably speak of lists and sets of variables. In programs (as well as descriptions of transformations), lists will often be used, whereas in semantic reasoning and calculation sets will be preferred. This choice can be justified by the fact that in our use of lists, the order of elements will only be significant for matching with corresponding lists (e.g. the two lists in a multiple assignment statement “\(x, y := 1, 2\)” and elements will not appear more than once (as is the case for sets).

Thus, we will also take the liberty to use set operations directly on (such) lists. This is a mere shorthand for taking the sets corresponding to those lists before applying the operation, and turning the result back into linear list form, afterwards.
The size of a set (or length of a list) \( V \) will be denoted \(|V|\).

### 3.1.2 Disjoint sets and tuples

Since disjointness of two sets will be used extensively, we adopt the notation \( V_1 \diamond V_2 \) as shorthand for \( V_1 \cap V_2 = \emptyset \) where \( V_1, V_2 \) are either lists or sets.

When referring to the union of disjoint sets (of variables), say \( X \) and \( Y \), we shall write \((X, Y)\). This should be understood as \( X \cup Y \) with an implicit statement that \( X \diamond Y \) is given. Note that as with \( n \)-tuples, any number of sets would be admitted, and the brackets are not optional.

That same notation shall be used also for pair (or in general \( n \)-tuple) forming. There, however, the elements will not necessarily be sets of variables.

Admittedly, having the same notation for both tuples and disjoint set-union is potentially confusing. Nonetheless, it appears that the actual meaning can be easily inferred from the context.

### 3.1.3 Generating fresh variable names

When performing a transformation, we shall soon find ourselves with a need to generate fresh variable names. For this purpose, we offer two versions of a function called \( \text{fresh} \). The first version shall take a pair \((n, V)\) as an argument, with \( n \) a natural number (for length) and \( V \) a set of variables (that are presently in use). In turn, it shall produce a set of fresh names, say \( X' \) such that (Q1:) \(|X'| = n\), and (Q2:) \( X' \diamond V \).

Here, (Q1:) and (Q2:) are names of the formal requirements (or postconditions). Those names will be recalled when applying \( \text{fresh} \), in hints of derivation steps. We shall use this format throughout the thesis.

Using an infix ‘\.' (= full stop) for function application, we shall be writing \( X' := \text{fresh}(n, V) \), where the \( n \) will typically be the length of a given set, say \(|X|\).

When new instances (e.g. for backup) of existing variables will be required, the second version of \( \text{fresh} \) will be used. It takes the form \( X' := \text{fresh}(X, V) \) with \((X, V)\) a pair of sets of variables. This time, we postulate (Q1:) \(|X'| = |X|\), (Q2:) \( X' \diamond V \), as before, and an extra requirement (Q3:) \( X = \text{sv}.X' \) where \( \text{sv} \) is a globally available mapping of such freshly generated variables to their corresponding original source variable.

### 3.2 Predicate calculus

#### 3.2.1 The state-space metaphor

In imperative programming, a given program, say \( S \), manipulates variables by changing their corresponding value. The collective value of all program variables, at any point of execution, is
known as ‘the state’. A computation under control of $S$ begins with a given initial state (i.e. its input), and terminates (if at all) in a final state (i.e. its output).

This terminology follows a metaphor of a ‘state space’, according to which, each program variable, say $x$, being associated with a possibly infinite but denumerable and non-empty set of distinct possible values (i.e. its type, denoted $T_x$), stands for a dimension of the state space. Each possible value, $val \in T_x$, is then associated with a single coordinate. Thus, any point in the state space uniquely represents (by its coordinates) the value of all program variables. (This should not be confused with abstract values, which are normally represented by program variables — it is the variables’ corresponding concrete values that are represented, at least metaphorically, by a point in the state space.)

### 3.2.2 Structures, expressions and predicates

According to DS, a structure is “an abstraction over expressions in program variables in the sense that the state space with its individually named dimensions has been eliminated from the picture” [13, Page 5]. There, that abstraction was chosen for developing a general theory. Since we adopt their theory only in the context of program semantics, we shall directly speak of expressions (over the state space). Note that structures, and hence expressions in program variables, are associated with a type. Thus integer expressions are distinguished from e.g. boolean expressions. The latter expressions are also known as predicates.

Hence, a predicate is an expression whose so-called global (i.e. free) variables are program variables. When evaluated, on a particular state, those variables are assigned (i.e. replaced with) the specific values (corresponding to that state). Thus, a predicate expresses a dichotomy on a program’s state space.

The syntax for expressing predicates includes the constants (or boolean scalars) false and true, relational operations (e.g. $=, \neq, <, \leq$) on expressions with program variables and possibly logical variables which must be local (i.e. bound) in the predicate, logical connectives (e.g. $\neg$, $\land$, $\lor$, $\equiv$), the universal and existential quantifiers ($\forall$ and $\exists$ respectively) and specific predicate transformers (i.e. functions from predicates to predicates) which will define the semantics of our programming language.

The universal ($\forall$) and existential ($\exists$) quantifiers generalise conjunction and disjunction, respectively. The format of the former is (quoted here from DS):

$$\forall \text{dummies : range : term} \ .$$

“Here, dummies stands for an unordered list of local variables, whose scope is delineated by the outer parenthesis pair. In what follows, $x$ and $y$ will be used to denote dummies; the dummies may be of any understood type.
The two components range and term are boolean structures, and so is the whole quantified expression, which is a boolean scalar if both range and term are boolean scalars. Range and term may depend on the dummies; their potential dependence on the dummies will be indicated explicitly by using a functional notation, e.g., if a range has the form \( r \times s \times y \), it is a conjunction of \( r \times x \) which may depend on \( x \) but does not depend on \( y \), and \( s \times x \times y \), which may depend on both.

...For the sake of brevity, the range true is omitted” [13, Pages 62-63].

The format for existential quantification is similar, only with \( \exists \) in place of the \( \forall \).

### 3.2.3 Square brackets: the ‘everywhere’ operator

As mentioned, predicates may involve global occurrences of program variables. An important function from boolean structures (i.e. predicates) to boolean scalars (i.e. true and false) is the so-called everywhere operator [13, Page 8]. Its application is denoted by surrounding a predicate, say \( P \), with a pair of square brackets, \([P]\).

When applied to a boolean scalar, the everywhere operator acts as identity (i.e. \([true] = true\) and \([false] = false\)); when applied to a predicate on a given state space, it acts as the universal quantification over all variables (i.e. dimensions) of that space ([13, Page 115]). Thus, \([P]\) yields true if and only if \( P \) holds in every single point of the state space (i.e. for any possible assignment of values to program variables occurring in it).

### 3.2.4 Functions and equality

Functions in DS are always total in their arguments (i.e. well-defined for all possible values of their arguments). They are defined as the unique solution of “an equation that contains the argument(s) of the function being defined as parameter(s)” [13, Page 18].

Function application — as mentioned, denoted by an infix ‘.’ (= full stop) — is left-associative, such that \( f \times y \) should be read as \((f \times x) \times y\).

For example, an integer increment function \( \text{incr} \times x = x + 1 \) (with the operator + itself defined as a function of its operands) is defined as the solution of \( y : [y = x + 1] \) or simply \([\text{incr} \times x = x + 1]\).

Note that the equality of a pair of expressions over the state space, as in \( y = x + 1 \), is not merely a boolean scalar, but rather a boolean expression over that same state space. (For example, consider the state space spanned by \( (x, y) \); there, applying \( y = x + 1 \) to point \((5, 6)\) yields \(true\) but applying it to \((5, 7)\) yields \(false\).) Hence, it is the square brackets, i.e. the everywhere operator, that turns the boolean expression into a scalar.

That function application preserves equality — a statement attributed to Leibniz and hence sometimes referred to, in hints, as “Leibniz” — is formulated, for any function \( f \) and arguments
CHAPTER 3. FORMAL SEMANTICS: PREDICATE TRANSFORMERS

\[ [x = y] \implies [f.x = f.y] \quad . \]  

(3.1)

The equality of expressions of type boolean, \( i.e. \) equivalence of predicates, can be written either with \( = \) or \( \equiv \). The latter is assigned a lower binding power than all other logical connectives, such that the round brackets in \( e.g. [(P \implies Q) \equiv (\neg P \lor Q)] \) can be removed.

3.2.5 Global variables in expressions, predicates and programs

The set of program variables occurring as global (\( i.e. \) free) variables in any expression \( E \) (including predicates) and program statement \( S \) will be referred to as \( \text{glob}.E \) and \( \text{glob}.S \), respectively.

For convenience and brevity, we shall allow the argument to \( \text{glob} \) to be any mixed \( n \)-tuple of expressions and statements. This should be read as shorthand for the union of all individual sets. For example \( \text{glob}.(S_1, P, E, S_2) \) is short for \( \text{glob}.S_1 \cup \text{glob}.P \cup \text{glob}.E \cup \text{glob}.S_2 \).

3.2.6 Substitutions

Functions from predicates to predicates, known as predicate transformers, will shortly be presented and (later) applied for defining program semantics. In our context of predicates on a state space, such primitive transformers are known as substitution predicate transformers [13, Page 114]. (See also [13, Chapter 2] for an introduction on substitution and replacement.)

A new predicate, say \( P' \), can be generated from an existing predicate \( P \), by replacing all global occurrences of program variables \( V \) with a matching list (in length and corresponding types) of expressions \( E \). For the syntax of substitutions we deviate from DS (who would write \( (V := E).P \) and adopt Morgan’s \( P[V \setminus E] \) (for \( P \) with \( V \) replaced by \( E \), [13]). Those square brackets are assigned the highest binding power; with postfix application being left associative, this will allow writing \( e.g. f.(P[V_1 \setminus E_1].Q[V_2 \setminus E_2][V_3 \setminus E_3]) \) for \( (f.(P[V_1 \setminus E_1]).((Q[V_2 \setminus E_2])[V_3 \setminus E_3])). \)

(Also, this format will cleanly allow later definition of special kinds of substitution, by prefixing \( V \) with the new substitution’s name.)

By definition, substitution distributes over all logical connectives. When distributing a substitution over a quantifier, potential name clashing (\( i.e. \) if local variables whose scope is bound by the quantifier have the same name as global variables in \( E \)) is avoided by renaming the local ones.

Just as we did with the function \( \text{glob} \) above, for convenience, and since predicates are merely boolean expressions in program variables, we apply substitutions to any expression, program statement, or a mixed \( n \)-tuple of those. For statements, we shall avoid potential problems (\( e.g. \) what does it mean to replace the target of an assignment with an expression?) by restricting ourselves to a so-called simple substitution [13, Page 105], substituting variables by variables.
Moreover, to avoid introducing aliases, the new variables will have to be distinct and fresh. More
precisely, for freshness in $S[X \setminus Y]$ we expect $Y \circ (\text{glob}.S \setminus X)$.

Some properties of substitution are worth noting. In the following, let $A$ stand for any expres-
sion (including predicates) or statement.

Let $X$ be any list of variables; then redundant self-substitutions can always be introduced or
removed; we thus postulate

$$A[X \setminus X] = A$$  \hspace{1cm} (3.2)

Another simplification allows the merge of following substitutions, if they form a needless chain;
as in the postulate

$$A[X \setminus Y][Y \setminus E] = A[X \setminus E]$$  \hspace{1cm} (3.3)

provided $Y \circ \text{glob}.A \setminus X$.

From the preceding two postulates, we can derive conditions for removing (or introducing)
redundant reversed double substitutions. Thus

$$A[X \setminus Y][Y \setminus X] = A$$  \hspace{1cm} (3.4)

provided $Y \circ \text{glob}.A \setminus X$.

A different kind of merge of following substitutions, is postulated for cases when substituted
variables are disjoint, and the first substitution does not affect the second. Thus

$$A[X1 \setminus E1][X2 \setminus E2] = A[X1, X2 \setminus E1, E2]$$  \hspace{1cm} (3.5)

provided $X1 \circ X2$ and $X2 \circ \text{glob}.E1$.

Finally, we can simply derive from the preceding postulate the conditions for swapping inde-
pendent substitutions. Thus

$$A[X1 \setminus E1][X2 \setminus E2] = A[X2 \setminus E2][X1 \setminus E1]$$  \hspace{1cm} (3.6)

provided $X1 \circ X2$, $X1 \circ \text{glob}.E2$ and $X2 \circ \text{glob}.E1$.

3.2.7 Proof format

According to the DS proof format [13, Chapter 4], designed for avoiding needless repetition in long
derivations, if $[A = C]$ can be proved by $[A = B]$ and $[B = C]$ for some intermediate expression
$B$, we write

$$A$$

$$= \quad \{\text{here comes a hint to why } [A = B]\}$$

$$B$$
from which the desired \([A = C]\) can be inferred. Note that \(A\), \(B\) and \(C\) are not necessarily boolean expressions, and hence the \(=\) rather than the more specific \(\equiv\). In any case, following DS, even for boolean expressions the \(=\) will be preferred (in derivations). The \(\equiv\), instead, will mostly be used in single line expressions, thus exploiting its low binding power and emphasising that the arguments are boolean.

Since \([A \Rightarrow B] \land [B \equiv C] \Rightarrow [A \Rightarrow C]\), when some steps in a derivation are of implication (\(\Rightarrow\)), the conclusion is an implication too. Similarly for the follows from (\(\Leftarrow\)) connective, as long as the two (\(\Rightarrow\) and \(\Leftarrow\)) are not mixed (in a single derivation).

### 3.2.8 From the calculus

The following set of theorems and equations are borrowed from the calculus of boolean structures, as defined (and proved) by Dijkstra and Scholten in [13]. Instead of an exhaustive collection, we state here only non-trivial results that will be of use in the course of this thesis.

The first theorem is proved in (DS: 5,96) of [13] — i.e. Equation 96 on Chapter 5).

**Theorem 3.1.** For any set \(W\), predicate \(P\), and any function \(f\) from the (type of the) elements of \(W\) to predicates

\[
W \neq \emptyset \Rightarrow [(\forall x : x \in W : P \land f.x) \equiv P \land (\forall x : x \in W : f.x)] ,
\]

i.e. , provided the range is non-empty, conjunction distributes over universal quantification.

The following is taken from (DS: 5,69) in [13].

**Theorem 3.2 (Contra-positive).** For any \(P, Q\)

\[
[P \Rightarrow Q \equiv \neg Q \Rightarrow \neg P] .
\]

The (punctual) monotonicity of quantifiers is borrowed from (DS: 5,102) and [13] Page 79.

**Theorem 3.3.** For any \(r, f, g\)

\[
[(\forall x : r.x \Rightarrow f.x) \Rightarrow ((\forall x : r.x \Rightarrow f.x) \equiv (\forall x : r.x : g.x))] \quad (3.9)
\]

\[
[(\exists x : r.x \Rightarrow f.x) \Rightarrow ((\exists x : r.x \Rightarrow f.x) \equiv (\exists x : r.x : g.x))] . \quad (3.10)
\]

Another property of existential quantification is taken from [13] Page 79.
Theorem 3.4. For any $P, r, f$

$$[P \land (\exists x : r.x : f.x) \equiv (\exists x : r.x : P \land f.x)] \quad (3.11)$$

Finally, the Laws of Absorption are proved in (DS: 5.23) and (DS: 5.24) of [13].

Theorem 3.5. Conjunction and disjunction satisfy the Laws of Absorption, i.e., for any $P, Q$

$$[P \land (P \lor Q) \equiv P] \quad (3.12)$$

$$[P \lor (P \land Q) \equiv P] \quad (3.13)$$

3.3 Program semantics

3.3.1 Predicate transformers

In Dijkstra and Scholten’s approach to program semantics, a program $S$ stands for the set of all computations possible under its control. With respect to a predicate $P$, defining a dichotomy on the state space on which $S$ operates, each computation $C$ may be of one of the following three classes: (1) “eternal” (i.e. fails to terminate); (2) “finally $P$” (i.e. terminates in a final state satisfying $P$); or (3) “finally $\neg P$” (i.e. terminates in a final state satisfying $\neg P$).

Each of the following three predicates defines a dichotomy on the initial state space of $S$ (with the second and third corresponding to final states satisfying $P$): (1) $wp.S.true$ (i.e. each computation under control of $S$ is either “finally $P$” or “finally $\neg P$”); (2) $wlp.S.P$ (i.e. either “eternal” or “finally $P$”); and (3) $wlp.S.(\neg P)$ (either “eternal” or “finally $\neg P$”).

Here $wlp.S$ emerges as a function from predicates to predicates, i.e. a predicate transformer, and $wlp$ stands for weakest liberal precondition. Only universally conjunctive predicate transformers are admitted as $wlp.S$. (See the following section for a formal definition of different types of junctivity.)

Similarly to $wlp$, the weakest precondition predicate transformer $wp.S$ is defined as $[wp.S.P \equiv wp.S.true \land wlp.S.P]$ for all $P$.

Being functions (from predicates to predicates), predicate transformers enjoy Leibniz’s Rule, i.e. for any predicate transformer $f$, we have $[P \equiv Q] \Rightarrow [f.P \equiv f.Q]$.

A predicate transformer is said to be monotonic (with respect to implication) if and only if $(\forall P, Q :: [P \Rightarrow Q] \Rightarrow [f.P \Rightarrow f.Q])$. Indeed, all predicate transformers used for program semantics will be monotonic.

3.3.2 Different types of junctivity

A predicate transformer $f$ is universally conjunctive if and only if

$$(\forall V : V \text{ is a bag of predicates} : [f.(\forall P : P \in V : P) \equiv (\forall P : P \in V : f.P)])$$

; similarly, it is
universally disjunctive if and only if
\[(\forall V : V \text{ is a bag of predicates} : [f.(\exists P : P \in V : P) \equiv (\exists P : P \in V : f.P)] \).

Other (weaker) types of junctivity include positive junctivity and finite junctivity. The former differs from universal junctivity in that its junctivity should apply not to any bag (of predicates) \(V\), but rather to non-empty ones, whereas the latter’s junctivity is expected to apply to any non-empty bag with a “finite number of distinct predicates” [13, Page 87].

From the definitions, it follows that if a predicate transformer \(f\) is universally conjunctive it is also positively so, and if positively conjunctive it is finitely so. It can also be shown that if \(f\) is finitely conjunctive, it is monotonic (as defined above). Finally, note that a similar weakening order holds for the corresponding disjunctivity types.

In the next chapter (see Section 4.1.5) we shall define the semantics of a programming language exclusively from universally disjunctive and positively conjunctive predicate transformers. It should be noted that all substitutions, defined earlier (in Section 3.2.6) as predicate transformers, are universally junctive (see [13, Page 117]).

As an example for the use of finite conjunctivity, consider the following theorem, which deals with absorption of termination.

**Theorem 3.6** (Absorption of Termination). For any statement \(S\) and predicate \(P\), we have

\[
[wp.S.P \land wp.S.true \equiv wp.S.P]
\]

provided \(wp.S\) is finitely conjunctive. We also have

\[
[wp.S.P \lor wp.S.true \equiv wp.S.true]
\]

provided \(wp.S\) is finitely disjunctive.

**Proof.** For the former, we observe

\[
wp.S.P \land wp.S.true
= \{ wp.S \text{ is finitely conjunctive (proviso)} \}
wp.S.(P \land true)
= \{ \text{identity element of } \land \} 
wp.S.P
,
\]

and then for the latter, we similarly observe

\[
wp.S.P \lor wp.S.true
= \{ wp.S \text{ is finitely disjunctive (proviso)} \}
wp.S.(P \lor true)
\]
Note that each part of the proof, being two steps long, will only save one step, whenever applied. However, at least the former case will be extensively used, and will thus worth its while.

Since our focus will be on deterministic programs, we shall now turn to define formally what is meant by a program being deterministic.

### 3.3.3 A definition of deterministic program statements

Interpreting the predicate $\text{wp}.S.(\neg P)$ as holding in all initial states for which no computation under control of $S$ is “finally $P$”, leads to another interesting predicate, $\neg\text{wlp}.S.(\neg P)$, holding in initial states for which there exists such a computation. So $\text{wp}.S.P$ holds where termination in $P$ is unavoidable and $\neg\text{wlp}.S.(\neg P)$ holds where terminating in $P$ is merely possible. Dijkstra and Scholten’s interpretation of a program being deterministic follows “what is possible is also unavoidable”, and thus the expectation $[\text{wp}.S.P \Rightarrow \neg\text{wlp}.S.(\neg P)]$ for all $P$.

Since it can be shown that $[\text{wp}.S.P \Rightarrow \neg\text{wlp}.S.(\neg P)]$ for all $P$, a program $S$ is considered deterministic if and only if

$$[\text{wp}.S.P \equiv \neg\text{wlp}.S.(\neg P)]$$

for all $P$.

The so-called conjugate of a predicate transformer $f$, is a predicate transformer, $f^*$, for which $[f^*.P \equiv \neg f.(\neg P)]$ holds for any predicate $P$. Surely, due to the redundancy of double negation, “if one predicate transformer is the conjugate of another, they are each other’s conjugate” [13, Page 83] (and hence the term conjugate).

Thus, the definition of deterministic statements can be rewritten, as (see also (DS: 7,7) in [13])

**Definition 3.7.** We have for any statement $S$

$$(S \text{ is deterministic}) \equiv (\text{wp}.S \text{ and } \text{wlp}.S \text{ are each other’s conjugate})$$

for all $P$.

The significance of this definition of conjugates comes from the fact that for any predicate transformer $f$ and its conjugate $f^*$ and all types of junctivity, we have (DS: 6,13)

$$(\text{the conjunctivity type of } f) = (\text{the disjunctivity type of } f^*)$$

for all $P$. 


3.4 Program refinement

In his PhD thesis [2], Back introduced in 1978 the concept of refinement as a binary relation between programs. Adapted to the DS notation, refinement can be formally defined as follows.

**Definition 3.8 (Refinement).** For any pair of program statements, \( S \) and \( T \), statement \( S \) is said to be refined by \( T \) (or \( T \) is a refinement of \( S \)), writing \( S \sqsubseteq T \), when for any predicate \( P \) we have \([wp.S.P \Rightarrow wp.T.P]\).

Since the semantics of the programming language is defined with monotonic predicate transformers, any part of a given program can be replaced with a refinement of itself, independently of the surrounding program.

In refinement calculi (e.g. [4, 45]), the programming language admits both specifications and executable constructs, known as code. The goal is a process for construction of provably correct code. According to the proposed process, this goal is achieved by specifying the requirements formally, using a so-called specification statement. Then, in a stepwise manner, the specification is refined into code. Each step is taken from a vocabulary of provably correct laws of refinement. We shall introduce a similar set of laws, as relevant for our context, in the next chapter.

Refinements have been shown to be useful for behaviour-preserving transformations of existing code (e.g. by Ward [56] and Cornéllo [11]). Ward applies refinements for e.g. reengineering and migration of code from one language to another [58]. Accordingly, his object language, a wide spectrum language (called WSL), is fundamentally non-deterministic.

Cornéllo has applied refinements directly in the context of refactoring [11] in his PhD thesis from 2004. There, a large number of known refactorings have been formulated for a Java-like object-oriented language, called ROOL, and applied in introducing known design patterns [23] into a given program.

We follow, in this thesis, a similar path of using the refinement relation in developing behaviour-preserving transformations. However, for simplicity, and since refactoring is concerned with transforming code, we restrict ourselves to a simple imperative deterministic language. Furthermore, instead of targeting a wide range of refactorings, we focus on the specific problem of slice extraction.

Slices have been formalised in the context of refinement by Ward [57, 59]. We return to those definitions later in the thesis (in Chapter 7).

This concludes our presentation of background to the formal semantics of this thesis. In summary, we have adopted Dijkstra and Scholten’s program semantics of predicate transformers. Set theory and the refinement relation have also been introduced, and will be used when manipulating programs and developing behaviour-preserving transformations.
Chapter 4

A Theoretical Framework

The original part of this thesis, as introduced so far, begins in this chapter, in which we develop a theoretical framework for proving correct transformations. The framework, building on traditional refinement calculus, will aim to support transformations of programs written in a simple imperative programming language. In contrast to earlier work on refinement, the language will be restricted to deterministic constructs — thus avoiding the need to synchronize duplicated non-deterministic choices, as will be explained. This decision is justified by the observation that refactoring is concerned with transforming existing code rather than specifications.

The chapter begins with a preliminary section, in which some basic concepts are defined. Then, a variation on Dijkstra’s language of guarded commands is introduced and formalised through weakest-preconditions semantics. This will be the object language for our transformations.

The framework will be extended and specialised later, e.g. with a slicing-based proof method in the next chapter and a slicing algorithm in Chapter 9, and will hence be applied in the formulation and development of solutions to slice extraction and related transformations.

4.1 Preliminaries

As a programming notation, this thesis adopts a subset of Dijkstra’s guarded commands, along with selected elements from Dijkstra and Scholten’s “Predicate Calculus and Program Semantics” (DS) [13]. As will be described shortly, a subset of Dijkstra’s language is chosen as our core language. This is then extended with some advanced constructs borrowed, with adaptations, from e.g. Morgan’s “Programming from Specifications” [45].

For the sake of simplicity, we choose to make some restricting assumptions on our programming language. These will allow a concise formulation of transformations. Admittedly, some of those assumptions are non-realistic and others might just not be desirable (e.g. due to performance
considerations). We return to discuss those choices in the concluding chapter, where we (briefly and informally) evaluate the applicability of the approach to modern programming languages. There, we shall propose to complement language extensions with extra applicability conditions.

Hence, in our language, all variables may be copied, leading to an independent clone in a new storage location. This includes the ability to clone the input and output streams. We restrict our attention to sequential programs, and expressions in our language (appearing in statements such as an assignment, or the guard of an IF statement) have no side-effects. Moreover, features such as aliasing, class hierarchy, overloading, exceptions or concurrency have been left out. As mentioned, possible implications of including such features are evaluated in the concluding chapter.

4.1.1 On slips and slides: an alternative to substatements

A slice captures a subset of the original behaviour, thus it is said to be a subprogram. When slicing is syntax preserving (as it normally is), one may also wish to say a slice (of statement $S$ with respect to variables $V$) is a substatement (of $S$). But is that so? And what is a substatement, anyway?

For example, let $S$ be the statement “if $x > y$ then $m := x$ else $m := y$ fi”; now let $S_1$ be “$m := x$ ” and $S_2$ be “if $x > y$ then $m := x$ fi”. Is $S_1$ a substatement of $S$? how about $S_2$?

Some may consider the former a substatement, since in terms of syntax trees, it may stand for a subtree. At the same time, others might claim the latter is a substatement, since in terms of nodes in a flow graph, it represents a subgraph.

We avoid such potential confusion, in this thesis, by refraining from speaking of substatements. Instead, $S_1$, being a subtree, is said to be a slip of $S$, whereas $S_2$ is a slide.

Any part of a statement which is in itself a statement is a slip (of that statement). Thus, if $S$ is a primitive statement, $S$ itself is its only slip. However, when $S$ is a compound statement (i.e. is compounded of parts $S.i$, each of which is a statement in itself), then $S$ itself is one of its slips, and all slips of each such $S.i$ are, too, slips (or even proper slips) of $S$. Those slips of each $S.i$ are sometimes referred to as proper slips, whereas the slips $S.i$ themselves are also considered immediate slips (of $S$). In terms of the abstract syntax tree (AST), a slip corresponds to a subtree.

A slide is complementary to a slip and is defined for each pair of statement and slip. The slide of $S$ on itself is the statement $S$ with all immediate slips (if any) replaced with the empty statement skip. When $S$ is a compound statement with immediate slips $S.i$, a slide of $S$ on a slip $T$ of $S.j$ is the statement $S$ with $S.j$ replaced with the slide of $S.j$ on $T$, and all other immediate slips $S.i$ (with $i \neq j$) replaced with the empty statement skip. In terms of the AST, a slide of $S$ on $T$ corresponds to the nodes on the path from (the node acting as root of) $S$ to (the root of its subtree) $T$. But slides will not be further discussed until later, in Chapter 8 where they will be formalised.
CHAPTER 4. A THEORETICAL FRAMEWORK

4.1.2 Why deterministic?

The main reason for focusing on deterministic programs is illustrated by the following inequivalence:

\[
\begin{align*}
\text{if true} & \rightarrow x,y := 1,1 \\
\text{□ true} & \rightarrow x,y := 2,2 \\
\text{fi}
\end{align*}
\]

\[
\begin{align*}
\text{if true} & \rightarrow x := 1 \\
\text{□ true} & \rightarrow x := 2 \\
\text{fi} \\
\text{if true} & \rightarrow y := 1 \\
\text{□ true} & \rightarrow y := 2 \\
\text{fi}
\end{align*}
\]

Whereas \( x = y \) is a true postcondition (for any initial state) of the program on the left, the other may terminate with \( e.g. \ x = 1 \land y = 2 \).

In essence, when duplicating a non-deterministic choice, one must synchronize the two choices in order to ensure behaviour preservation.

In this thesis we avoid such cases by restricting ourselves to deterministic programs. Future extension of this work to include non-determinism should be possible and interesting.

Our decision can also be justified, as mentioned earlier, by the observation that non-determinism is typically useful in specification and during design process whereas refactoring is concerned with transforming actual code.

4.1.3 On deterministic program semantics

In general, \( \text{wlp}.S \) is more fundamental than \( \text{wp}.S \) as there is no way of defining the former in terms of the latter. However, for deterministic programs, we observe that due to \((3.16)\) and the redundancy of double negation (twice) \( \text{wlp}.S \) can be defined by

\[
[\text{wlp}.S.Q \equiv \neg\text{wp}.S.(\neg Q)]
\]

Thus in this thesis we leave weakest liberal preconditions alone and define the semantics of our programming language solely in terms of weakest preconditions.

But before dismissing \( \text{wlp} \), we shall use it once more, in investigating the difference between refinement and equivalence of deterministic program statements. (A final mention of \( \text{wlp} \) will follow, in Section 4.1.5)

4.1.4 On refinement, termination and program equivalence

Two deterministic program statements \( S, T \) are considered semantically equivalent if for all \( P \), we have \([\text{wp}.S.P \equiv \text{wp}.T.P]\). In this thesis this is denoted \( S = T \). A weaker relation is that of
refinement. There, $S$ is said to be refined by $T$ (or $T$ is a refinement of $S$, denoted $S \sqsubseteq T$) if 
$[wp.S.P \Rightarrow wp.T.P]$ for all $P$.

Essentially, what is the difference between the two relations? Clearly, it can be shown (through
predicate calculus) that $S = T$ if and only if $S \sqsubseteq T \land S \sqsupseteq T$. But what does it mean for $T$
to be a refinement of $S$ and $S$ not a refinement of $T$ (and thus $S \neq T$)? This happens when $T$
is “more terminating” than $S$. Operationally speaking, $T$ may terminate on input for which $S$
does not. But on input for which both terminate, the final state is guaranteed to be the same. In other
words, if $S$ is refined by $T$ and both terminate under the exact same conditions, they are also
equivalent. This is formulated in the following theorem.

**Theorem 4.1.** Let $S, T$ be any two deterministic statements; then

$$(S = T) \equiv (S \sqsubseteq T \land [wp.S.true \equiv wp.T.true])$$

*Proof.*

\begin{align*}
S = T \\
= & \quad \text{(def. of program equivalence)} \\
(\forall P :: [wp.S.P \equiv wp.T.P]) \\
= & \quad \text{Lemma 4.2 (see below)} \\
(\forall P :: [wp.S.P \Rightarrow wp.T.P] \land [wp.S.true \equiv wp.T.true]) \\
= & \quad \text{pred. calc. (3.7: the range is non-empty)} \\
(\forall P :: [wp.S.P \Rightarrow wp.T.P]) \land [wp.S.true \equiv wp.T.true] \\
= & \quad \text{(def. of refinement)} \\
S \sqsubseteq T \land [wp.S.true \equiv wp.T.true].
\end{align*}

\[ \square \]

**Lemma 4.2.** Let $S, T$ be any two deterministic statements; then

$$(\forall P :: [wp.S.P \equiv wp.T.P]) \equiv (\forall P :: [wp.S.P \Rightarrow wp.T.P] \land [wp.S.true \equiv wp.T.true])$$

*Proof.* The LHS $\Rightarrow$ RHS part is trivial, due to predicate calculus ($\equiv$ implies $\Rightarrow$).

For LHS $\Leftarrow$ RHS, note that since $[wp.S.P \Rightarrow wp.T.P]$ is already given for any predicate $P$
(RHS), we only need to show $[wp.S.P \Leftarrow wp.T.P]$, for which we observe

\begin{align*}
wp.S.P \\
= & \quad \text{(def. of wp)} \\
wlp.S.P \land wp.S.true
\end{align*}
In contrast to refinement, program equivalence is amenable for deriving correct transformations in both directions. However, on one of those, a refinement may yield more accurate results (as it does in slicing by removing irrelevant loops even if those may not terminate — recall Section 2.2.2).

The above result is important as it allows us to confidently focus on developing refinement rules, where appropriate, knowing that the extra step of turning them into equivalences is always available.

4.1.5 Semantic language requirements

Dijkstra and Scholten insist on two basic requirements the semantics of each language construct must satisfy. Firstly (R0:) any \( wlp.S \) is universally conjunctive; and secondly (R1:) \( [wp.S.false \equiv false] \) for any \( S \). Requirement R1 — known as “The Law of the Excluded Miracle” — is due to the observation that no state satisfies \( false \) and the predicate \( wp.S.false \) holds in states where no computation under control of \( S \) exists.

When defining semantics in terms of weakest preconditions alone, requirement R0 can be replaced with a new requirement (RE1:) that \( wp.S \) is universally disjunctive. We prove that RE1 implies (for deterministic \( S \)) both R0 and R1 in what follows.

**Theorem 4.3.** Let \( S \) be any deterministic statement with (RE1:) \( wp.S \) being universally disjunctive; we then have both (R0:) \( wlp.S \) is universally conjunctive, and (R1:) \( [wp.S.false \equiv false] \).

**Proof.** First, for R0, we observe (on the lines of the proof for (DS: 7,9) in [13])

\[
\text{the conjunctivity type of } wlp.S \\
= \quad \{ \text{properties of conjugate: see } 3.18 \} 
\]
the disjunctivity type of \((\text{wlp}.S)^*\)

\[= \{\text{(3.17): } S \text{ is deterministic}\}\]

the disjunctivity type of \(\text{wp}.S\)

\[= \{\text{RE1}\}\]

universal .

Then, we observe for R1

\[
\text{wp}.S.\text{false}
\]

\[= \{\text{existential quantification over the empty range yields } false\}\]

\[
\text{wp}.S.(\exists P : P \in \emptyset : P)
\]

\[= \{\text{wp}.S \text{ is univ. disj. (RE1)}\}\]

\[
(\exists P : P \in \emptyset : \text{wp}.S.P)
\]

\[= \{\text{again, existential quantification over the empty range yields } false\}\]

false .

Thus, in our context, RE1 faithfully takes the place of DS’s R0,R1. We note that a consequence of RE1 and Theorem 4.3 above (thus having R0,R1 available) is that \(\text{wp}.S\) is positively conjunctive (as proved in (DS: 7,8) of [13]). However, it is not universally so for (possibly) non-terminating \(S\), since universal quantification over the empty range yields true.

4.1.6 Global variables in transformed predicates

According to our adopted formalism, programs manipulate predicates over the program’s state space, expressed syntactically as boolean structures. In our analysis and manipulation of such programs, we shall be interested in the set of global variables actually mentioned in the transformed predicates.

Let \(P\) be any predicate. We denote the set of global (i.e. free) variables in \(P\) as \(\text{glob}.P\). Let \(S\) be any given statement; variables in \(\text{glob}.P\) may be subject to direct substitution by the transformer \(\text{wp}.S\). What do we know of \(\text{glob}.(\text{wp}.S.P)\)?

First, we denote variables that will definitely be substituted by \(\text{wp}.S\) as \(\text{ddef}.S\). Those will be the variables that are definitely (i.e. for any initial state) defined by \(S\). (Examples of \(\text{ddef}\), as well as the other properties to be introduced shortly, will be given in Section 4.2.) Second, we observe that some of those variables, as well as others, may find their way into \(\text{glob}.(\text{wp}.S.P)\) even if not
in $\text{glob}.P$. Those are the variables whose initial value may affect the result of $S$ and are hence denoted as $\text{input}.S$. We are now ready to postulate requirement $\text{RE}2$:

$$\text{glob}.(\text{wp}.S.P) \subseteq ((\text{glob}.P \setminus \text{ddef}.S) \cup \text{input}.S)$$

for all $S, P$.

The transformation of $\text{wp}.S$ on some predicates will be restricted to adding a conjunct expressing termination of $S$. That is, for such $S, P$ we expect $[\text{wp}.S.P \equiv P \land \text{wp}.S.true]$. This is so whenever all variables in $\text{glob}.P$ are guaranteed not to be modified by $S$ (in a case where $S$ terminates). We thus denote by $\text{def}.S$ the set of variables that may be modified by $S$. That is, a variable $x$ must be in $\text{def}.S$ if there exists a terminating computation under control of $S$ for which the final value of $x$ differs from its initial value. When a variable is not in $\text{def}.S$ we know its initial and final values will in any case be the same. We hence postulate the requirement $\text{RE}3$:

$$[\text{wp}.S.P \equiv P \land \text{wp}.S.true]$$

for all $S, P$ with $\text{glob}.P \circ \text{def}.S$.

As can be expected, all definitely defined variables $\text{ddef}.S$ shall always take part in the set of (possibly) defined variables, $\text{def}.S$. We thus postulate the requirement $\text{RE}4$:

$$\text{ddef}.S \subseteq \text{def}.S$$

for all $S$.

In addition to the sets $\text{def}.S$, $\text{ddef}.S$ and $\text{input}.S$, we shall define for each language construct its set of global (i.e. free) variables. In fact, we shall expect this set to consist of variables from the three previously mentioned sets. Keeping in mind $\text{ddef}.S \subseteq \text{def}.S$ for all $S$ ($\text{RE}4$ above), we now postulate the requirement $\text{RE}5$:

$$\text{glob}.S = \text{def}.S \cup \text{input}.S$$

for all $S$. (Recall the overloading of $\text{glob}$, as was first mentioned in Section 3.2.5, being applicable for predicates, as before, as well as for program statements, as in this case, or even for program expressions of any type.)

Here is a summary of the required properties. For any statement $S$ we require

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{RE}1$</td>
<td>$\text{wp}.S$ is universally disjunctive</td>
</tr>
<tr>
<td>$\text{RE}2$</td>
<td>$\text{glob}.(\text{wp}.S.P) \subseteq ((\text{glob}.P \setminus \text{ddef}.S) \cup \text{input}.S)$ for all $P$</td>
</tr>
<tr>
<td>$\text{RE}3$</td>
<td>$[\text{wp}.S.P \equiv P \land \text{wp}.S.true]$ for all $P$ with $\text{glob}.P \circ \text{ddef}.S$</td>
</tr>
<tr>
<td>$\text{RE}4$</td>
<td>$\text{ddef}.S \subseteq \text{def}.S$</td>
</tr>
<tr>
<td>$\text{RE}5$</td>
<td>$\text{glob}.S = \text{def}.S \cup \text{input}.S$</td>
</tr>
</tbody>
</table>
4.2 The programming language

Here is an introduction to the chosen language constructs and their corresponding semantics. A full definition with proof of all requirements can be found in Appendix A.

4.2.1 Expressions, variables and types

As our transformations deal exclusively with statements and names of variables, while preserving all types (of variables and expressions), it is tempting and not uncommon to avoid any mention of those. However, to prevent confusion, it is worth mentioning the types with which our programming language (and hence the code examples in this thesis) shall be concerned.

The basic types (of variables and expressions) shall include integers (with typical arithmetic and relational operators) and booleans (with similar syntax to predicates).

Further to that, we shall allow variables of type array or stream. Each array variable, say \( a \), will always be associated with an extra variable, \( a.length \). A stream, dedicated either for reading (i.e. input) or writing (i.e. output), shall be implemented as a special case of array, with an extra (implicit) index variable associated with it. This variable shall be pointing to the next available location.

4.2.2 Core language

Assignment

The first language construct is the assignment statement. It takes the form \( X := E \) with \( X \) standing for a finite list of variables and with \( E \) standing for a list of expressions (of the same length as \( X \)). Type compatibility is assumed. This is a so-called simultaneous assignment (or multiple assignment), in which all expressions are evaluated before being assigned to their corresponding target variables. It should be noted that the target variables must be distinct. To the case where the lists \( X \) and \( E \) are both empty, we sometimes refer as “skip”.

\[
\begin{align*}
\text{[wp.]} & \quad X := E \quad P \equiv P[X \setminus E] \text{ for all } P;
\text{def.} & \quad X := E \quad \triangleq X;
\text{ddef.} & \quad X := E \quad \triangleq X;
\text{input.} & \quad X := E \quad \triangleq \text{glob}.E \quad \text{and}
\text{glob.} & \quad X := E \quad \triangleq X \cup \text{glob}.E.
\end{align*}
\]

It is worth noting that, for simplicity, all expressions in our language are assumed to be well formed and all operators and functions are complete and hence well defined for all possible values.

The special case of assignment to an array element, say \( \text{a}[i] := E \) shall be understood as an assignment to the whole array, \( \text{a} := \text{a}[i \mapsto E] \), meaning that the array \( \text{a} \) ends up being as
before in all elements other than the i-th, in which it gets the value of E.

An output stream, say out (with dedicated index variable, say out.i), can be appended through a statement “out << E”, which should be interpreted as “out, out.i := out[out.i := E], out.i + 1”. Similarly, reading from an input stream, say in, (with index variable in.i), takes the form “in >> x”, and should be interpreted as “x, in.i := in[in.i := x], in.i + 1”.

Sequential composition

The first compound construct is sequential composition. It takes the form “S1 ; S2” and starts executing S2 only after normal completion of S1.

\[ wp."S1 ; S2".P \equiv wp.S1.(wp.S2.P) \] for all P ;
\[ def."S1 ; S2" \triangleq def.S1 \cup def.S2 ; \]
\[ ddef."S1 ; S2" \triangleq ddef.S1 \cup ddef.S2 ; \]
\[ input."S1 ; S2" \triangleq input.S1 \cup (input.S2 \setminus ddef.S1) ; \] and
\[ glob."S1 ; S2" \triangleq glob.(S1, S2) . \]

Recall glob.(S1, S2) is short for glob.S1 \cup glob.S2.

Alternative construct

The alternative construct takes the form of “if B then S1 else S2 fi” (and is sometimes abbreviated to IF). Upon execution, if the guard B, a boolean expression, is evaluated to true, S1 is executed; otherwise, S2 is executed.

\[ wp.IF.P \equiv (B \Rightarrow wp.S1.P) \land (\neg B \Rightarrow wp.S2.P) \] for all P ;
\[ def.IF \triangleq def.S1 \cup def.S2 ; \]
\[ ddef.IF \triangleq ddef.S1 \cap ddef.S2 ; \]
\[ input.IF \triangleq glob.B \cup input.S1 \cup input.S2 ; \] and
\[ glob.IF \triangleq glob.B \cup glob.S1 \cup glob.S2 . \]

Repetitive construct

The repetitive construct takes the form of “while B do S od” (and is sometimes abbreviated to DO). Upon execution, if the guard B is evaluated to true, the guarded S is executed. Once S terminates successfully, the process is repeated, until the guard is evaluated to false, in which case the loop terminates successfully.
CHAPTER 4. A THEORETICAL FRAMEWORK

\[ wp.DO.P \equiv (\exists i : 0 \leq i : (k^i.true)) \] for all \( P \),
with \( k \) given by (DS:9,44) \[13\]: \( [k.Q \equiv (B \lor P) \land (\neg B \lor wp.S.Q)] \) ;
def.DO \( \triangleq \) def.S ;
ddef.DO \( \triangleq \emptyset \) ;
input.DO \( \triangleq \) glob.B \( \cup \) input.S ; and
glob.DO \( \triangleq \) glob.B \( \cup \) glob.S .

As for earlier language constructs (with the exception of substitutions), this formulation of wp.DO follows the DS notation — for its deterministic subset. Note, however, that the semantics of repetition could have been equally defined in terms of implication, by \( [k.Q \equiv (B \Rightarrow wp.S.Q)] \). This would render the similarity to the semantics of IF clearer: one could think of the DO statement as a recursive construct, say DO’, comprising an IF statement on the lines of “ if \( B \) then \( S \); DO’ else skip fi ”. Nevertheless, this thesis adopts the DS formulation of loops.

This completes our core language, a subset of Dijkstra and Scholten’s guarded commands \[13\]. The following constructs are extensions borrowed from Morgan \[45\], with some adaptations as our context requires. Later, in order to emphasise that a certain statement \( S \) is restricted to constructs of the core language, we shall call it a core statement.

4.2.3 Extended language

Assertions

An assertion statement (called “assumption” by Morgan \[45\]), is a boolean expression on the program state. If true, execution goes on normally; otherwise the program aborts. (In guarded commands, “the operational interpretation of abort is that for all initial states its execution fails to terminate”. \[13\] Page 135)

\[ [wp." \{ B \} " .P \equiv B \land P] \] for all \( P \) ;
def." \{ B \} " \( \triangleq \emptyset \) ;
ddef." \{ B \} " \( \triangleq \emptyset \) ;
input." \{ B \} " \( \triangleq \) glob.B ; and
glob." \{ B \} " \( \triangleq \) glob.B .

Assertions, locally expressing the surrounding context, will serve as a vehicle for performing correct local transformations and refinements. The assertions will typically be added to a program, temporarily, by propagating knowledge through a given (compound) statement. They will thus express intermediate as well as final results of a provably correct program analysis, prior to some transformation.
Local variables

Normally, in (block-based, imperative) programming languages, local variables serve for storing temporary results of computations, before using those in further computations. Being local to a certain statement, assignments to such variables bear no effect on the surrounding context (even if a variable with the same name exists there as well). According to Morgan’s definition, a local variable is initialised (on entry to its scope) to any possible value. However, since such non-determinism is not permitted in our context, we could either insist local variables must not be used before being defined — as is commonly enforced in modern languages — or agree on some initial value. As will be explained shortly, we prefer the latter.

A special case of local variables is that of parameters. Typically, a method (or procedure) may declare e.g. ‘value’ parameters; such local variables will be initialised, whenever the method is called, with a copy of the actual value sent. Again, any local modifications will be hidden from the caller. In [45], Morgan allows value parameters to be sent to any given statement, say $S$, through so-called value substitutions; e.g. “$S[\text{value } f \backslash E]$” would send the value of $E$ to locals $f$ in $S$.

It turns out that we do not require, in our context, the full power of such value substitutions. Instead, we shall do with what can be termed a self value substitution (i.e. “$S[\text{value } f \backslash f]$”, to use Morgan’s syntax). The effect of such self-substitution is that, on the one hand, any modification to $f$ in $S$ shall be local (i.e. hidden from the surrounding context), while, on the other hand, $f$ will be initialised to its actual value in that global context.

Since only self value substitutions will be needed, in this work, and since the value-substitution notation, for those, present a redundancy (i.e. repeating $f$ in the above), we opt to avoid the introduction of value substitutions. Instead, we shall get the same effect (of self value sub.) by assuming local variables are initialised to their corresponding global value. In reality, such initialisation should only take place if a local may be used before being defined.

Local variables may be introduced anywhere a statement is expected. Their introduction takes the form of “[[var $L$ ; $S$]]” where $S$ is any statement and $L$ stands for a list of variable names. If a local variable is used before being defined in $S$, its entry value is used. Since definitions of $L$ in $S$ should be local, its entry value is kept in a fresh backup variable on entry and retrieved on exit; accordingly, the semantic definition of “[[var $L$ ; $S$]]” follows that of “$L' := L; S; L := L'$” where $L'$ is fresh.
[wp." [var L ; S] " ] P ≡ (wp.S.P[L \ L']|[L' \ L]) for all P with glob.P \ L'; or the simpler
[wp." [var L ; S] " ] Q ≡ wp.S.Q] for all Q with glob.Q \ (L, L')
def." [var L ; S] " ≜ def.S \ L ;
ddef." [var L ; S] " ≜ ddef.S \ L ;
input." [var L ; S] " ≜ input.S ; and
glob." [var L ; S] " ≜ (def.S \ L) \ input.S; or
glob." [var L ; S] " ≜ (glob.S \ (L \ input.S)) .

Note that generality is not lost by restricting P as we do. Whenever elements of L' appear in the
postcondition, those can be locally renamed.

A common case is one in which the declared variables are immediately defined, e.g. " [var L ; L := E ; S]] " . In such cases, the shorthand " [var L := E ; S]] " is allowed.

**Live variables**

In program analysis \cite{17}, a variable x is considered live, at a given program point, if there exists
a path (from the given point) to a use of x, which is free of re-definitions of x. If a variable is not
live at a given point (i.e. it is dead), its value at that point is of no interest to the program, and
can be modified to anything.

In our refactorings we shall take advantage of this notion of liveness, for example in removing
dead assignments (i.e. assignments to dead variables). However, since there is no simple deter-
ministic way of saying “that variable can hold any value, at this point”, we choose to add the
concept of liveness to the programming language, or rather to the meta-language.

We do so by defining a dual of local variables. Instead of stating which variables are local to
a statement S, we explicitly state the variables that are not. This way, it can be assumed that
those variables will be live on exit. In contrast, all other variables, being local, will be guaranteed
to hold, on exit from S, their corresponding initial value. Thus, all local definitions of those will
be of no relevance to the surrounding context. Consequently, modifying those to any value, just
before exiting S, will bear no effect.

We define “ S[live V] " ≜ " [var L ; S] " where L := def.S \ V. Thus, the semantics and
properties can be derived from those of local variables, as is summarised in the following. For a
given statement S, set of variables V, a corresponding set L := def.S \ V and fresh L', we have:
4.3 Laws of program analysis and manipulation

The weakest-preconditions semantics, as defined above for all language constructs, along with known theorems from the predicate calculus, can and will be applied in proving a collection of laws for correct program analysis and manipulation.

Such laws, in turn, will be useful for proving and deriving rules of program equivalence, refinement and transformation. (The latter is distinguished from the former two in that instead of relating two programs, it shall describe how to produce a new program from a given one.)

The collection is by no means exhaustive, though; only laws to be directly useful in the thesis are formulated. A summary of those laws can be found in Appendix F, and all proofs are given in Appendix B.

4.3.1 Manipulating core statements

A first set of laws, designates easy manipulation of core statements. Very similar laws have been defined elsewhere (e.g. [45, 3, 56, 30]).

For example, the following is a definition of a law (see Law 3 in the appendices) to support the distribution of a statement into (or out of, when applied from right-to-left) both branches of a following IF statement, provided the former does not define (i.e. modify the value of) any variable that is tested in the IF’s guard:

Let $S, S_1, S_2, B$ be three statements and a boolean expression, respectively; then

\[
\text{" } S \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \text{" } = \text{" } S \text{ if } B \text{ then } S_1 \text{ else } S \text{ fi } \text{" }
\]

provided def.\,S \circ glob.\,B.

Another code-motion related law (Law 5 in the appendices), supports moving a (certain kind of loop-invariant) assignment statement forward, outside a DO loop’s body (or into its end, when applied from right-to-left):

\[
\text{def. } S[\text{live } V] \text{" } \triangleq \text{def. } S \cap V \\
\text{ddef. } S[\text{live } V] \text{" } \triangleq \text{ddef. } S \cap V \\
\text{input. } S[\text{live } V] \text{" } \triangleq \text{input. } S \\
\text{glob. } S[\text{live } V] \text{" } \triangleq (\text{def. } S \cap V) \cup \text{input. } S
\]
Let $S_1, X, B, E$ be any statement, set of variables, boolean expression and set of expressions, respectively; then

\[
\{ X = E \} \text{ while } B \text{ do } S_1 \text{ od } = \{ X = E \} \text{ while } B \text{ do } S_1 \text{ od }
\]

provided $X \circ (\text{glob}.B \cup \text{input}.S_1 \cup \text{glob}.E)$.

### 4.3.2 Assertion-based program analysis

When flow-sensitive properties of specific program points are desired, we shall introduce and then propagate assertions throughout the program, thus expressing both intermediate and final results of a program analysis. Again, similar sets of laws have been defined and employed elsewhere, e.g. by Back [2], Morgan [45] and Ward [56].

For example, the following (Law 7) supports both introduction and elimination of assertions, following an assignment:

Let $X, Y, E_1, E_2$ be two sets of variables and two sets of expressions, respectively; then

\[
X, Y := E_1, E_2 = X, Y := E_1, E_2 ; \{ Y = E_2 \}
\]

provided $(X, Y) \circ \text{glob}.E_2$.

The following law (Law 12) will be used for propagating assertions forward into branches of an IF statement, as well as backward ahead of the IF:

Let $S_1, S_2, B_1, B_2$ be two statements and two boolean expressions, respectively; then

\[
\{ B_1 \} \text{ if } B_2 \text{ then } S_1 \text{ else } S_2 \text{ fi } = \text{ if } B_2 \text{ then } \{ B_1 \} \text{ ; } S_1 \text{ else } \{ B_1 \} \text{ ; } S_2 \text{ fi }
\]

Note that this law is a direct corollary of the more general Law 3 (from above) and the fact that $\text{def}$ of assertions is empty.

After introducing and propagating assertions, and before eliminating them, they will typically be used in making substitutions. If two variables are known to hold the same value ahead of a statement, the immediate use of one can be replaced with the other, in that statement. By immediate use, we refer to the used expressions in assignments and the guard of an IF statement. In the guard of a DO loop, however, we can make such a substitution only if the required assertion is available both before the loop and at the end of its body. We refer to such substitutions, in hints, as assertion-based substitution.

Since such substitutions will often be preceded by an introduction of the assertion, following an assignment statement, we introduce a combined law (Law 18) to which we refer in hints as assignment-based substitution:
Let $S_1, S_2, B$ be two statements and a boolean expression, respectively; let $X, X', Y, Z,$
$E_1, E_1', E_2, E_3$ be four lists of variables and corresponding lists of expressions; then

“ $X, Y := E_1, E_2 ; Z := E_3$ ” = “ $X, Y := E_1, E_2 ; Z := E_3[Y \ E_2]$ ” ;

“ $X, Y := E_1, E_2 ; IF$ ” = “ $X, Y := E_1, E_2 ; IF'$ ” ; and

“ $X, Y := E_1, E_2 ; DO$ ” = “ $X, Y := E_1, E_2 ; DO'$ ”

provided $((X \cup X'), Y) \circ glob.E2$

where $IF := “ \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } ”$,

$IF' := “ \text{if } B[Y \ E_2] \text{ then } S_1 \text{ else } S_2 \text{ fi } ”$,

$DO := “ \text{while } B \text{ do } S_1 ; X', Y := E_1', E_2 \text{ od } ”$

and $DO' := “ \text{while } B[Y \ E_2] \text{ do } S_1 ; X', Y := E_1', E_2 \text{ od } ”$ .

4.3.3 Manipulating liveness information

Let $S, V$ be any statement and set of variables, respectively, and recall our definition of liveness
information, in our extended language. The fact that out of the variables defined in $S$, only those in
$V$ are live on exit from $S$ is expressed as “ $S[\text{live } V]$ ”. This is syntactic sugar for $[\text{var } coV ; S]$ where $coV := \text{def}.S \setminus V$.

Laws for manipulating liveness information (see Sections B.3 and F.3 for proofs of all laws and
summary, respectively) include introduction and removal of auxiliary information, distribution and
propagation of liveness information, and finally introduction and elimination of dead assignments.
(By auxiliary information we refer to information that is locally redundant but may have some
importance in the global context.)

Whenever only (a subset of the) mentioned live variables are actually defined (in a statement
$S$), the liveness information is redundant, and can be dropped. Conversely, any superset of the
defined variables (in any $S$) can safely augment $S$ as liveness information. This is expressed by
the following law (Law 19) for introducing and removing auxiliary liveness information:

Let $S, V$ be any statement and set of variables, respectively, with $\text{def}.S \subseteq V$; then

$S = “ S[\text{live } V] ”$ .

In propagating liveness information over sequential composition, it is interesting to see that,
on the one hand, some live-on-exit variables may be intermediately dead, whereas on the other
hand, some dead-on-exit variables may become intermediately live. This is demonstrated by the
set $V'$ in Law 20.
Let $S_1, S_2, V_1, V_2$ be any two statements and two sets of variables, respectively; then

\[
\text{“} (S_1 ; S_2)[live \ V_1] \text{“} = \text{“} (S_1[live \ V_2] ; S_2[live \ V_1])[live \ V_1] \text{“}
\]

provided $V_2 = (V_1 \setminus ddef.S_2) \cup input.S_2$.

Here, variables in $ddef.S_2$ are said to be “killed” by $S_2$ whereas variables in $input.S_2$ are “generated”. Such propagation of information, by removing KILL sets and adding GEN sets, is common practice in intraprocedural data flow analysis (see e.g. [47, Section 2.1]).

Note that we propagate information directly on the abstract syntax (i.e. its tree-like representation) rather than on a flow graph. This is made possible due to the simplicity and structured nature of our language. With that respect, it should also be noted that in all analyses, assuming the availability of the $def, ddef, input$ sets for all program elements, our algorithms will involve a single pass of the program’s tree. However, in presentation, we shall not be concerned with time or space complexities.

Another law for propagating liveness information is Law 22.

Let $B, S, V_1, V_2$ be any boolean expression, statement and two sets of variables, respectively; then

\[
\text{“} (\text{while} \ B \text{ do } S \text{ od})[live \ V_1] \text{“} = \text{“} (\text{while} \ B \text{ do } S[live \ V_2] \text{ od})[live \ V_1] \text{“}
\]

provided $V_2 = V_1 \cup (glob.B \cup input.S)$.

Liveness information, explicitly added to (and propagated through) a program, can help in identifying dead assignments. Those can subsequently be removed. The following is one such law for dead-assignment elimination (Law 24):

Let $S, V, Y, E$ be any statement, two sets of variables and set of expressions, respectively; then

\[
\text{“} S[live \ V] \text{“} = \text{“} (S \ ; \ Y := E)[live \ V] \text{“}
\]

provided $Y \circ V$.

Note that the law can (and indeed will) also be used to introduce dead assignments.

### 4.4 Summary

This completes the initial introduction to our framework for refactoring. A subset of Dijkstra’s language of guarded commands, along with extensions for representing program analysis information have been defined with predicate transformer semantics and related sets of variables. Those have been used in presenting laws for program analysis and manipulation.

The next chapter will extend our framework by applying some of its elements in devising a novel method for proving the correctness of slicing-based refactoring transformations.
Chapter 5

Proof Method for Correct Slicing-Based Refactoring

Our framework for slicing-based refactoring is enhanced in this chapter, with the development of a proof method for the refinement of deterministic statements. This method will be specifically tailored for slicing-based refactoring.

5.1 Introducing slice-refinements and co-slice-refinements

A law of refinement typically associates two meta-programs, \( S \) and \( T \), with some applicability conditions. Any two programs satisfying those conditions are then guaranteed to be related through refinement; \( S \) is then said to be refined by \( T \), such that the latter preserves the total correctness of the former. Operationally speaking, whenever both versions are started in the same state, one in which \( S \) is known to be terminating, we expect \( T \) to produce the exact same result as \( S \) (i.e. to terminate in the same state). With input for which \( S \) does not terminate, \( T \) is allowed to do anything.

Taken formally, in terms of predicate-transformer semantics, we expect that for any given predicate \( P \), the weakest precondition of \( T \) applied to \( P \) will follow from that of \( S \) on (the same) \( P \), everywhere. Thus, when aiming to prove the correctness of such a refinement law, we are required to show \([\text{wp}.S.P \Rightarrow \text{wp}.T.P]\) for all \( P \).

Alternatively, when restricting ourselves to deterministic statements, one can prove the correctness of a refinement law in a slice-wise manner. That is, instead of considering general predicates over the full state space, we consider predicates over a slice (corresponding to a subset of the program variables, or accordingly to the subspace spanned by their potential values) separately from predicates over its complement.
We first define a relation of slice-refinement and a complementary relation of co-slice-refinement as follows.

\[(S \subseteq_V T) \equiv (\forall P : \text{glob}.P \subseteq V : [\text{wp}.S.P \Rightarrow \text{wp}.T.P]),\]

in which case \(T\) is said to be a slice-refinement of \(S\) with respect to \(V\); and

\[(S \subseteq_{(\diamond V)} T) \equiv (\forall P : \text{glob}.P \diamond V : [\text{wp}.S.P \Rightarrow \text{wp}.T.P]),\]

in which case \(T\) is said to be a co-slice-refinement of \(S\) with respect to \(V\).

The subscript \(V\) in \(S \subseteq_V T\) means (as the definition shows) the refinement relation holds for all predicates with global variables in \(V\). Accordingly, the \((\diamond V)\) in \(S \subseteq_{(\diamond V)} T\) guarantees the refinement holds for all predicates with no global mention of \(V\).

### 5.2 Variable-wise proofs

With the above definitions, we now investigate how each slice-refinement (as well as co-slice-refinements, later) can be proved in a point-wise fashion. Instead of considering any possible postcondition (on the sliced variables \(V\)), only very particular postconditions in the form of \(x = \text{val}\), for any variable \(x \in V\) and possible value \(\text{val}\), are considered.

In the following, we assume any variable, \(x\), has a type, \(T\cdot x\), associated with it (even if that type is not explicitly declared in the program). All variable types are assumed to be non-empty, possibly infinite, sets of distinct values.

#### 5.2.1 Proving slice-refinements

**Theorem 5.1.** For any pair of deterministic statements \(S\) and \(SV\) and any set of variables \(V\), we have

\[(S \subseteq_V SV) \equiv (\forall x, val : x \in V \land val \in T \cdot x : [\text{wp}.S.(x = val) \Rightarrow \text{wp}.SV.(x = val)]).\]

Here, the (otherwise arbitrary) name \(SV\) was chosen as a hint that this statement has something to do with \(S\) and \(V\).

Before proving the theorem, we turn to some motivation. At first glance, the theorem may seem obvious, perhaps due to our mention of point-wise proof and the universal disjunctivity of \(\text{wp}.S\) and \(\text{wp}.T\). However, a closer look reveals that this alternative view of slice-refinement is more variable-wise than it is point-wise (even though the proof will indeed involve points in the state space).

Furthermore, it turns out that this formulation is not (always) suitable in the presence of non-determinism. Recall the example from Section 4.1.2 where our focus on deterministic programs was justified. Despite the fact that all postconditions of the form \("x = val"\) or \("y = val"\) yielded \(\text{false}\) as weakest precondition of both programs, the duplicated version was not a slice-refinement.
of the other one with respect to \( V = \{x, y\} \). This was revealed by the postcondition \( x = y \) and the fact that \( \text{true} \Rightarrow \text{false} \).

We are now ready for a proof of correctness, keeping in mind that \( S, SV \) are both deterministic programs and hence \( \text{wp}.S, \text{wp}.SV \) are both universally disjunctive and positively conjunctive.

**Proof.** (LHS \( \Rightarrow \) RHS): Trivial; \( \text{glob}.\text{true} = \emptyset \) and \( \text{glob}.(x = \text{val}) \subseteq V \) for all \( x \in V \) of type \( T \), and value \( \text{val} \in T \times V \).

(LHS \( \Leftarrow \) RHS): We need to prove that for any predicate \( P \) with \( \text{glob}.P \subseteq V \) we have \( [\text{wp}.S.P \Rightarrow \text{wp}.SV.P] \).

The only two predicates on the empty space (when \( V = \emptyset \)), are the boolean scalars \( \text{false} \) and \( \text{true} \). Now \([\text{wp}.S.\text{false} \Rightarrow \text{wp}.SV.\text{false}]\) is given by the Law of Excluded Miracle, and \([\text{wp}.S.\text{true} \Rightarrow \text{wp}.SV.\text{true}]\) is given by the (RHS) proviso.

Thus in the remainder of this proof we shall assume \( V \) is not empty. We now note that the predicate \( P \) expresses a dichotomy on the state subspace spanned by variables \( V \). This dichotomy can also be represented by the (possibly infinite) set of points at which the predicate is evaluated to \( \text{true} \). Each point can then be represented (by its coordinates) as a conjunction of simple formulae of the form \( x = \text{val} \), one formula for each variable (axis) in \( V \).

Let \( n := |V| \) and let \( p \) enumerate all \( n \)-dimensional points in the state space \( (p.i \text{ is the value of the } i \text{-th dimension, } i.e. \text{ of variable } V.i) \); and \( P.p \) is \( \text{true} \) if \( P \) (with a substitution of each variable, \( V.i \), with its corresponding value \( p.i \)) is evaluated to \( \text{true} \) at \( p \); then \( P \) can be rewritten as:

\[
[P \equiv (\exists p : P.p = \text{true} : (\forall i : 0 \leq i < n : V.i = p.i))] \quad . \tag{5.1}
\]

We now need to prove that \([\text{wp}.S.P \Rightarrow \text{wp}.SV.P]\) (for all \( P \) with \( \text{glob}.P \subseteq V \)) under the assumptions \([\text{wp}.S.\text{true} \Rightarrow \text{wp}.SV.\text{true}]\) and for all \( x \in V \) and value \( \text{val} \in T \times V \) we have:

\[
[\text{wp}.S.(x = \text{val})] \Rightarrow [\text{wp}.SV.(x = \text{val})] \quad . \tag{5.2}
\]

We recall \( V \) is non-empty \( (i.e. \ 0 < n) \) and observe (for all \( P \) with \( \text{glob}.P \subseteq V \))

\[
\begin{align*}
\text{wp}.S.P & = (\text{5.1} \text{ above and Leibniz}) \\
& = \{\text{RE1: wp}.S \text{ is universally disjunctive}\} \\
& = (\exists p : P.p = \text{true} : \text{wp}.S.(\forall i : 0 \leq i < n : V.i = p.i)) \\
& = \{0 < n \text{ and wp}.S \text{ is positively conjunctive (S deterministic)}\} \\
& = (\exists p : P.p = \text{true} : (\forall i : 0 \leq i < n : \text{wp}.S.(V.i = p.i))) \\
& \Rightarrow \{\text{assumption 5.2 above}\}
\end{align*}
\]
\[(\exists p : P.p = true : (\forall i : 0 \leq i < n : wp.SV.(V.i = p.i)))\]

\[=\{0 < n \text{ and } wp.SV \text{ is positively conjunctive } (SV \text{ deterministic})\}\]

\[(\exists p : P.p = true : wp.SV.(\forall i : 0 \leq i < n : V.i = p.i))\]

\[=\{\text{RE1: } wp.SV \text{ is universally disjunctive}\}\]

\[wp.SV.(\exists p : P.p = true : wp.SV.(\forall i : 0 \leq i < n : V.i = p.i))\]

\[=\{\text{above and Leibniz}\}\]

\[wp.SV.P.\]

Note the correctness of the \((\Rightarrow)\) step above, due to the monotonicity of both \(\forall \) \((3.9)\) and \(\exists \) \((3.10)\).

\[\square\]

5.2.2 A co-slice-refinement is a slice-refinement of the complement

Co-slice-refinements, as slice-refinements, can be proved in a variable-wise manner.

**Corollary 5.2.** For any pair of deterministic statements \(S\) and \(ScoV\) and any set of variables \(V\), we have

\[(S \subseteq_{c(V)} ScoV) \equiv ([wp.S.true \Rightarrow wp.ScoV.true] \land
(\forall x, val : x \in coV \land val \in T, x, wp.S.(x = val) \Rightarrow wp.ScoV.(x = val))))\]

where \(coV := ((\text{def}S \cup \text{def}ScoV) \setminus V)\).

**Proof.** Recalling \(S\) and \(ScoV\) are deterministic, we observe

\[(S \subseteq_{c(V)} ScoV)\]

\[=\{\text{Theorem 5.3 see below}\}\]

\[(S \subseteq_{co} ScoV)\]

\[=\{\text{Theorem 5.1 with } SV, V := ScoV, coV\}\]

\[([\text{def}S.true \Rightarrow wp.ScoV.true] \land\]

\[(\forall x, val : x \in coV \land val \in T, x \in [wp.S.(x = val) \Rightarrow wp.ScoV.(x = val)])\] .

\[\square\]

**Theorem 5.3.** A co-slice-refinement is a slice-refinement of the complementary set of defined variables. That is, for any pair of deterministic statements \(S\) and \(ScoV\) and any set of variables \(V\), we have

\[(S \subseteq_{c(V)} ScoV) \equiv (S \subseteq_{co} ScoV)\]

where \(coV := ((\text{def}S \cup \text{def}ScoV) \setminus V)\).
Proof. (LHS ⇒ RHS): Due to $V \odot coV$, all predicates $P$ with $glob.P \subseteq coV$ (as required on the RHS) have $glob.P \odot V$. Thus the LHS yields $[wp.S.P \Rightarrow wp.ScOV.P]$.

(LHS ⇐ RHS): We observe for all $P$ with $glob.P \odot V$ and $glob.P \setminus coV \neq \emptyset$ (without the latter the RHS would already yield the required $[wp.S.P \Rightarrow wp.ScOV.P]$):

$$wp.S.P = \{\text{pointwise version of } P \text{ on the state space spanned by } (coV1, ND):$$

let $coV1 := glob.P \cap coV$, $ND := glob.P \setminus coV$, $n := |coV1|$ and $n' := |glob.P|$}$$

$$wp.S.(\exists p : P.p = true : (\forall i : 0 \leq i < n : coV1.i = p.i) \wedge$$

$$\exists i : n \leq i < n' : ND.(i - n) = p.i))$$

$$\Rightarrow \{\text{junctivity of } wp.S: \text{recall } S \text{ is deterministic and }$$

$$0 < |ND| \text{ due to } (glob.P \setminus coV) \neq \emptyset\}$$

$$wp.S.(\exists p : P.p = true : wp.S.(\forall i : 0 \leq i < n : coV1.i = p.i) \wedge$$

$$\forall i : n \leq i < n' : wp.S.true \wedge (ND.(i - n) = p.i)))$$

$$\Rightarrow \{\text{RE3: } ND \odot def.S \text{ and RE2}\}$$

$$wp.ScoV.((\exists p : P.p = true : wp.ScOV.(\forall i : 0 \leq i < n : coV1.i = p.i) \wedge$$

$$\forall i : n \leq i < n' : wp.ScOV.true \wedge (ND.(i - n) = p.i)))$$

$$\Rightarrow \{\text{RE3: } ND \odot def.ScOV \text{ and RE2}\}$$

$$wp.ScOV.((\exists p : P.p = true : wp.ScOV.(\forall i : 0 \leq i < n : coV1.i = p.i) \wedge$$

$$\forall i : n \leq i < n' : wp.ScOV.(ND.(i - n) = p.i)))$$

$$\Rightarrow \{\text{junctivity of } wp.ScOV: ScOV \text{ is deterministic and again } 0 < |ND|\}$$

$$wp.ScOV.P = \{\text{pointwise version of } P\}$$
5.3 Slice and co-slice refinements yield a general refinement

Combining separate variable-wise proofs for a slice-refinement and its complementary co-slice-refinement, we can discard the variable-wise approach.

**Corollary 5.4.** Let $S$, $T$ be any pair of deterministic statements and let $V$ be any set of variables; then

$$(S \subseteq T) \equiv ((S \subseteq V \ T) \land (S \subseteq_{(\circ V)} T))$$

**Proof.** We observe

- $S \subseteq T$
  
  - $\{\text{def. of refinement; glob}.P \circ \emptyset \text{ holds for all } P\}$
  
  - $(\forall P : \text{glob}.P \circ \emptyset : [\text{wp}.S.P \Rightarrow \text{wp}.T.P])$

- $S \subseteq_{(\circ V)} T$
  
  - $\{\text{Corollary 5.2 with ScoV}, V := T, \emptyset; S, T \text{ deterministic}\}$
  
  - $[\text{wp}.S.\text{true} \Rightarrow \text{wp}.T.\text{true}] \land$
    
    - $(\forall x, \text{val} : x \in (\text{def.S} \cup \text{def.T}) \land \text{val} \in T.x : [\text{wp}.S.(x = \text{val}) \Rightarrow \text{wp}.T.(x = \text{val})])$

- $[\text{Lemma 5.5 (see below) and pred. calc.}]$
  
  - $[\text{wp}.S.\text{true} \Rightarrow \text{wp}.T.\text{true}] \land$
    
    - $(\forall x, \text{val} : x \in (\text{def.S} \cup \text{def.T}) \land \text{val} \in T.x : [\text{wp}.S.(x = \text{val}) \Rightarrow \text{wp}.T.(x = \text{val})] \land$
    
    - $(\forall x, \text{val} : x \in (V \setminus (\text{def.S} \cup \text{def.T})) \land \text{val} \in T.x :$
      
      - $[\text{wp}.S.(x = \text{val}) \Rightarrow \text{wp}.T.(x = \text{val})])$

- $[\text{merging the ranges}]$
  
  - $[\text{wp}.S.\text{true} \Rightarrow \text{wp}.T.\text{true}] \land$
    
    - $(\forall x, \text{val} : x \in (V \cup \text{def.S} \cup \text{def.T}) \land \text{val} \in T.x :$
      
      - $[\text{wp}.S.(x = \text{val}) \Rightarrow \text{wp}.T.(x = \text{val})])$

- $[\text{splitting the range; pred. calc.}]$
  
  - $[\text{wp}.S.\text{true} \Rightarrow \text{wp}.T.\text{true}] \land$
    
    - $(\forall x, \text{val} : x \in V \land \text{val} \in T.x : [\text{wp}.S.(x = \text{val}) \Rightarrow \text{wp}.T.(x = \text{val})] \land$
    
    - $[\text{wp}.S.\text{true} \Rightarrow \text{wp}.T.\text{true}] \land$
      
      - $(\forall x, \text{val} : x \in ((\text{def.S} \cup \text{def.T}) \setminus V) \land \text{val} \in T.x :$
        
        - $[\text{wp}.S.(x = \text{val}) \Rightarrow \text{wp}.T.(x = \text{val})])$

- $[\text{Theorem 5.1 with SV := T and Corollary 5.2 with ScoV := T: again,}$
  
  - $S, T \text{ are deterministic}]$
Lemma 5.5. Let $S, T$ be any pair of statements and let $P$ be any predicate, with $\text{glob}.P \circ (\text{def}.S \cup \text{def}.T)$; then

$$[\text{wp}.S.true \Rightarrow \text{wp}.T.true] \Rightarrow [\text{wp}.S \Rightarrow \text{wp}.T].$$

Proof. For all such $S, T, P$, with $[\text{wp}.S.true \Rightarrow \text{wp}.T.true]$ and $\text{glob}.P \circ (\text{def}.S \cup \text{def}.T)$, we observe

$\text{wp}.S.P$

$= \{\text{RE3}: \text{glob}.P \circ \text{def}.S \text{ (proviso and set theory)}\}$

$P \land \text{wp}.S.true$

$\Rightarrow \{\text{proviso}\}$

$P \land \text{wp}.T.true$

$= \{\text{RE3 again: } \text{glob}.P \circ \text{def}.T \text{ (proviso and set theory)}\}$

$\text{wp}.T.P$. 

\[ \square \]

5.3.1 A corollary for program equivalence

An immediate corollary of the above refinement proof method will support proof of program equivalence.

Corollary 5.6. Let $S, T$ be any pair of deterministic statements and let $V$ be any set of variables; then

$$(S = T) \equiv$$

$$(\forall P : \text{glob}.P \subseteq V : [\text{wp}.S.P \equiv \text{wp}.T.P]) \land (\forall Q : \text{glob}.Q \circ V : [\text{wp}.S.Q \equiv \text{wp}.T.Q])$$.

Proof. Recalling $S$ and $T$ are deterministic, we observe

$S = T$

$= \{\text{Theorem 4.1}\}$

$(S \subseteq T) \land [\text{wp}.S.true \equiv \text{wp}.T.true]$

$= \{\text{Corollary 5.4}\}$

$(S \subseteq V T) \land (S \subseteq (\circ V) T) \land [\text{wp}.S.true \equiv \text{wp}.T.true]$

$= \{\text{def. of slice-refinement and co-slice-refinement}\}$
5.4 Example proof: swap independent statements

To illustrate our new method of proof, consider the following program equivalence for swapping independent statements:

**Program equivalence 5.7.** Let \( S_1, S_2 \) be any pair of deterministic statements; then

\[
“ S_1 ; S_2 ” = “ S_2 ; S_1 ”
\]

provided \( \text{def}.S_1 \circ \text{def}.S_2, \text{def}.S_1 \circ \text{input}.S_2 \) and \( \text{input}.S_1 \circ \text{def}.S_2 \).

Note that the provisos are actually gathered from the following derivation. This is representative of our general approach to refinement, program equivalence and transformation.

**Proof.** We first observe for all \( P \) with \( \text{glob}.P \subseteq \text{def}.S_1 \) (note that \( \text{def}.S_2 \circ \text{glob}.P \) due to proviso \( \text{def}.S_1 \circ \text{def}.S_2 \)):

\[
\begin{align*}
\text{wp.} “ S_1 ; S_2 ”.P &= \{ \text{wp of } “ ; “ \} \\
\text{wp}.S_1.\text{wp}.S_2.P &= \{ \text{RE3: def}.S_2 \circ \text{glob}.P \} \\
\text{wp}.S_1.(P \land \text{wp}.S_2.\text{true}) &= \{ \text{conj. of wp}.S_1 \} \\
\text{wp}.S_1.P \land \text{wp}.S_1.\text{true} \land \text{wp}.S_2.\text{true} &= \{ \text{RE3: def}.S_1 \circ \text{glob}.(\text{wp}.S_2.\text{true}) \text{ due to RE2 and proviso def}.S_1 \circ \text{input}.S_2 \} \\
\text{wp}.S_1.P \land \text{wp}.S_1.\text{true} \land \text{wp}.S_2.\text{true} &= \{ \text{absorb term. (3.14)} \}
\end{align*}
\]
\[ \text{wp.} S_1.P \land \text{wp.} S_2.\text{true} \]
\[ = \{ \text{RE3: def.} S_2 \circ \text{glob.}(\text{wp.} S_1.\text{P}) \text{ due to RE2, proviso input.} S_1 \circ \text{def.} S_2, \]
\[ \text{and choice of } P \} \]
\[ \text{wp.} S_2.(\text{wp.} S_1.\text{P}) \]
\[ = \{ \text{wp of } \cdot ; \cdot \} \]
\[ \text{wp.}^{\text{absorb term.}} S_2 ; S_1 \text{ "}, P . \]

We now observe for all \( P \) with \( \text{glob.} P \circ \text{def.} S_1 \):
\[ \text{wp.}^{\text{absorb term.}} S_2 ; S_1 \text{ "}, P \]
\[ = \{ \text{wp of } \cdot ; \cdot \} \]
\[ \text{wp.} S_2.(\text{wp.} S_1.\text{P}) \]
\[ = \{ \text{RE3: def.} S_1 \circ \text{glob.} P \} \]
\[ \text{wp.} S_2.(P \land \text{wp.} S_1.\text{true}) \]
\[ = \{ \text{conj. of wp.} S_2 \} \]
\[ \text{wp.} S_2.P \land \text{wp.} S_2.(\text{wp.} S_1.\text{true}) \]
\[ = \{ \text{RE3: def.} S_2 \circ \text{glob.}(\text{wp.} S_1.\text{true}) \text{ due to RE2 and proviso input.} S_1 \circ \text{def.} S_2 \} \]
\[ \text{wp.} S_2.P \land \text{wp.} S_2.\text{true} \land \text{wp.} S_1.\text{true} \]
\[ = \{ \text{absorb term. (3.14)} \} \]
\[ \text{wp.} S_2.P \land \text{wp.} S_1.\text{true} \]
\[ = \{ \text{RE3: def.} S_1 \circ \text{glob.}(\text{wp.} S_2.\text{P}) \text{ due to RE2, proviso def.} S_1 \circ \text{input.} S_2, \]
\[ \text{and choice of } P \} \]
\[ \text{wp.} S_1.(\text{wp.} S_2.P) \]
\[ = \{ \text{wp of } \cdot ; \cdot \} \]
\[ \text{wp.}^{\text{absorb term.}} S_1 ; S_2 \text{ "}, P . \]

Taken together, the above two derivations yield the required program equivalence, due to Corollary \[5.6\] and the determinism of \( S_1 \) and \( S_2 \).

\[\square\]

### 5.5 Summary

This chapter has extended our transformation framework by introducing a proof method for both refinements and program equivalence, specifically designed to support slicing-related refactoring transformations. Two complementary concepts of slice-refinement and co-slice-refinement have
been introduced. It has been shown that proving each kind of refinement separately is equivalent to proving normal refinements of code. This approach has been shown to be applicable for proving program equivalence as well, and such an example, for swapping independent statements, has been proven.

The next chapter will apply this proof method in developing our first version of sliding.
Chapter 6

Statement Duplication

In this chapter, the first step towards slice extraction is taken by formally developing a program equivalence that yields a transformation of statement duplication. The duplication begins by making two clones of the original program. These are composed sequentially, and correctness is ensured by the addition of compensatory code. This code is responsible for keeping and retrieving backup of initial and final values. One clone is specialized for computing the results carried by the variables selected for extraction, whereas the other is dedicated to the remaining computations (as captured by the remaining variables).

6.1 Example

When asked to extract the computation of $sum$ in the following program fragment

```plaintext
while i<a.length do
  i,sum,prod :=
  i+1,sum+a[i],prod*a[i]
  od
```

we offer to duplicate the selected statement, and systematically add some compensatory code, to make the transformation correct. This would yield the following version, in which the actual computation of $sum$ is in the first clone, whereas the remaining results (i.e. in $i$, $prod$) are computed in the second (complementary) clone:
6.2 Sequential simulation of independent parallel execution

Consider the effects of a program statement’s execution as the results carried by its defined variables, i.e. def.S for a statement S, in case of termination. Now consider a partition of def.S, say to two subsets def.S = (V, coV). (Here, the otherwise arbitrary name coV was chosen as a hint that this set of variables is complementary to V.)

If S is deterministic, its computation can be accomplished by two — or more, depending on the number of partitions — independent machines. Each such machine will be given the same initial state and the same program statement for execution. Clearly, due to determinism, both machines terminate under the same conditions. Then, in case of termination, the results can be collected from the two machines, say V from the first machine and coV from the second.

The usefulness of the above construction will become clear later on, when each clone of S will be independently simplified to achieve its specific designated goal.

For simulating the above scenario (of two independent machines), using our sequential imperative language (for a single machine), we propose to sequentially compose two clones of statement S. For the second clone to work properly, we insist that the first clone will not modify the original set of variables. This will be guaranteed by keeping a backup of initial values, and retrieving those values upon entry to the second clone. Using our available language constructs, this transformation is formalised in the following section. (We actually formalise it as a program equivalence, thus keeping it general enough to be relevant for the reverse transformation as well.)

6.3 Formal derivation

Program equivalence 6.1. Let S, V, coV, iV, icoV, fV be any deterministic statement and five sets of variables, respectively; then

\[
\begin{align*}
\text{[var isum,iprod,ii,fsum} \\
\text{; isum,iprod,ii := sum,prod,i} \\
\text{; while i<a.length do} \\
\text{ i,sum,prod :=} \\
\text{ i+1,sum+a[i],prod*a[i]} \\
\text{ od} \\
\text{ ; fsum := sum}
\end{align*}
\]

\[
\begin{align*}
\text{sum,prod,i := isum,iprod,ii} \\
\text{; while i<a.length do} \\
\text{ i,sum,prod :=} \\
\text{ i+1,sum+a[i],prod*a[i]} \\
\text{ od} \\
\text{ ; sum:=fsum}
\end{align*}
\]
provided \( \text{def}.S = (V, coV) \)
and \( (iV, icoV, fV) \odot \text{glob}.S \).

Proof.

\[
S = \begin{cases}
   "(iV, icoV := V, coV) \quad V, coV := iV, icoV \\
   ; S \quad ; S \\
   ; fV := V \quad ; V := fV)[live V, coV] 
\end{cases}
\]

\[
\text{[prepare for statement duplication (Lemma 6.2 below):]}
\]
\[
\text{def}.S = (V, coV) \quad \text{and} \quad (iV, icoV, fV) \odot \text{glob}.S
\]

\[
"(iV, icoV := V, coV \quad S \quad fV := V \quad V := fV)[live V, coV] "
\]

\[
= \begin{cases}
   \text{[statement duplication (Lemma 6.3 below):} \quad S \text{ is deterministic]} \\
   "(iV, icoV := V, coV \quad S \quad fV := V \quad V, coV := iV, icoV \quad S \quad V := fV)[live V, coV] " 
\end{cases}
\]

\[\square\]

Lemma 6.2. Let \( S, V, coV, iV, icoV, fV \) be any statement and five sets of variables, respectively; then

\[
S = \begin{cases}
   "(iV, icoV := V, coV) \quad V := fV)[live V, coV] "
\end{cases}
\]

provided \( \text{def}.S = (V, coV) \)
and \( (iV, icoV, fV) \odot \text{glob}.S \).

Proof.

\[
"(iV, icoV := V, coV \quad S \quad fV := V \quad V := fV)[live V, coV] "
\]

\[
= \begin{cases}
   \text{[assignment-based sub. (Law 18):} \quad V \odot fV \quad \text{since} \quad V \subseteq \text{def}.S \quad \text{(proviso),} \quad \text{def}.S \subseteq \text{glob}.S \quad \text{(RE5) and} \quad \text{glob}.S \odot fV \quad \text{(proviso)}] \\
   "(iV, icoV := V, coV \quad S \quad fV := V \quad V := fV)[live V, coV] "
\end{cases}
\]

\[
= \begin{cases}
   \text{[remove aux. self assignment (Law 2)]}
   "(iV, icoV := V, coV \quad S \quad fV := V)[live V, coV] "
\end{cases}
\]
\[ = \{ \text{remove dead assignments (Law 24): } fV \odot (V, coV) \text{ (proviso and RE5)} \} \]

\[ " (iV, icoV := V, coV ; S)[live V, coV] " \]

\[ = \{ \text{remove dead assignments (Law 25): } (iV, icoV) \odot ((V, coV) \setminus ddef.S) \cup input.S) \text{ (again, proviso and RE5)} \} \]

\[ " S[live V, coV] " \]

\[ = \{ \text{remove aux. liveness info. (Law 19): } def.S \subseteq (V, coV) \} \]

\[ S. \]

**Lemma 6.3.** Let \( S, V, coV, iV, icoV, fV \) be any deterministic statement and five sets of variables, respectively; then

\[
\begin{array}{ccc}
" iV, icoV := V, coV " & = & " V, coV := iV, icoV " \\
; S & ; S & ; S \\
; fV := V & ; fV := V & ; fV := V \\
\end{array}
\]

provided \( def.S = (V, coV) \)

and \( (iV, icoV, fV) \odot glob.S . \)

**Proof.**

\[ " iV, icoV := V, coV ; S ; fV := V " \]

\[ = \{ \text{intro. following assertion (Law 7)} \} \]

\[ " iV, icoV := V, coV ; \{ V, coV = iV, icoV \} ; S ; fV := V " \]

\[ = \{ \text{see below} \} \]

\[ " iV, icoV := V, coV ; \{ V, coV = iV, icoV \} ; S ; fV := V \]

\[ ; V, coV := iV, icoV ; S " \]

\[ = \{ \text{remove following assertion (Law 7)} \} \]

\[ " iV, icoV := V, coV ; S ; fV := V ; V, coV := iV, icoV ; S " . \]

Note that no data may flow from the first clone to the second:

\( def." S ; fV := V " \odot input." V, coV := iV, icoV ; S " . \) We now observe for all \( P \)

\[ wp." \{ V, coV = iV, icoV \} ; S ; fV := V ; V, coV := iV, icoV ; S " . P \]

\[ = \{ wp \text{ of } ; \text{ and assertions} \} \]

\[ (V, coV = iV, icoV) \wedge wp." S ; fV := V ; V, coV := iV, icoV ; S " . P \]
= \{ \text{wp of } '; \text{ and } ' :=' } \}
= (V, coV = iV, icoV) \land \text{wp}.S.(\text{wp}. " V, coV := iV, icoV ; S ".P)\{fV \setminus V\}
= \{ \text{wp of } '; \text{ and } ' :=' } \}
= (V, coV = iV, icoV) \land \text{wp}.S.(\text{wp}.S.P)[V, coV \setminus iV, icoV][fV \setminus V] .

At this point, due to Corollary [5.6] and the determinism of S, we are ready to distinguish two complementary cases: (a) \( \text{glob}.P \subseteq fV \); and (b) \( \text{glob}.P \circ fV \). The former case involves results computed in the first clone of S whereas the latter takes care of the computations from the second clone, which are — due to the lack of data flow — independent of the first clone’s results.

**Case (a):** \( \text{glob}.P \subseteq fV \)

\[
(V, coV = iV, icoV) \land \text{wp}.S.(\text{wp}.S.P)[V, coV \setminus iV, icoV][fV \setminus V]
= \{ \text{RE3: } \text{glob}.P \circ \text{def}.S} \}
= (V, coV = iV, icoV) \land \text{wp}.S.(\text{wp}.S.true \land P)[V, coV \setminus iV, icoV][fV \setminus V] \land P[V, coV \setminus iV, icoV][fV \setminus V] \}
= \{ \text{remove redundant subs: } \text{RE2 (} fV \circ (\text{input}.S, iV, icoV)) \}
= (V, coV = iV, icoV) \land \text{wp}.S.(\text{wp}.S.true)[V, coV \setminus iV, icoV] \land P[V, coV \setminus iV, icoV][fV \setminus V] \}
= \{ \text{remove redundant subs: } (V, coV) \circ \text{glob}.P \}
= (V, coV = iV, icoV) \land \text{wp}.S.(\text{wp}.S.true)[V, coV \setminus iV, icoV] \land P[fV \setminus V] \}
= \{ \text{wp of } \text{conj.} \}
= (V, coV = iV, icoV) \land \text{wp}.S.(\text{wp}.S.true)[V, coV \setminus iV, icoV] \land \text{wp}.S.P[fV \setminus V] \}
= \{ \text{RE3: } (iV, icoV) \circ \text{def}.S \text{ (recall } (V, coV) = \text{def}.S) \}
= (V, coV = iV, icoV) \land \text{wp}.S.true \land (\text{wp}.S.true)[V, coV \setminus iV, icoV] \land \text{wp}.S.P[fV \setminus V] \}
= \{ \text{remove redundant subs: } (V, coV = iV, icoV) \}
= (V, coV = iV, icoV) \land \text{wp}.S.true \land \text{wp}.S.true \land \text{wp}.S.P[fV \setminus V] \}
= \{ \text{absorb termination [3.14], twice} \}
= (V, coV = iV, icoV) \land \text{wp}.S.P[fV \setminus V] \}
= \{ \text{wp of } ' :=' \text{ and } ',' \}
= (V, coV = iV, icoV) \land \text{wp}. " \text{S ; } fV := V ".P
= \{\text{wp of assertions and} \ i \ ; \ i\} \\
\text{wp}.^* \{iV, \ico V = V, \co V\} \ ; S \ ; fV \assign V \ Wy.P.

\textbf{Case (b):} \ \glob.P \circ fV

\begin{align*}
(V, \co V = iV, \ico V) \land \wp.S.((\wp.S.P)[V, \co V \setminus iV, \ico V][fV \setminus V]]
\end{align*}

= \{\text{remove redundant sub.:} \ fV \circ (\glob.S \cup \glob.P \cup (iV, \ico V)), \text{ proviso and RE2}\}

\begin{align*}
(V, \co V = iV, \ico V) \land \wp.S.((\wp.S.P)[V, \co V \setminus iV, \ico V] \\
\end{align*}

= \{\text{RE3:} \ \glob.((V, \co V := iV, \ico V).((\wp.S.P)) \circ \text{def.S}}

\begin{align*}
&\text{(recall \ (V, \co V) = \text{def.S})} \\
(V, \co V = iV, \ico V) \land \wp.S.\text{true} \land (\wp.S.P)[V, \co V \setminus iV, \ico V] \\
\end{align*}

= \{\text{remove redundant subs:} \ (V, \co V = iV, \ico V)\}

\begin{align*}
(V, \co V = iV, \ico V) \land \wp.S.\text{true} \land \wp.S.P \\
\end{align*}

= \{\text{absorb termination 3.14}\}

\begin{align*}
(V, \co V = iV, \ico V) \land \wp.S.P \\
\end{align*}

= \{\text{intro. redundant sub.: proviso}\}

\begin{align*}
(V, \co V = iV, \ico V) \land \wp.S.P[fV \setminus V] \\
\end{align*}

= \{\text{wp of} \ i := \ i \ ; \ i\} \\

\begin{align*}
(V, \co V = iV, \ico V) \land \wp. \{iV, \ico V = V, \co V\} \ ; S \ ; fV \assign V \ Wy.P. \\
\end{align*}

\textbf{6.4 Summary and discussion}

This chapter has introduced our first solution to slice extraction, through a naive sliding approach of statement duplication. Two clones of a given statement are composed for sequential execution. The first clone, \textit{i.e.} the extracted code, is dedicated for computing a selected subset of the original program’s results, whereas the second clone, \textit{i.e.} the complement, is responsible for the remaining results. Behaviour preservation is guaranteed, as is formally proved in the chapter, by the addition of compensatory code.

This includes copying of the initial state, saving it in backup variables, in an approach borrowed from the Tuck transformation of Lakhotia and Deprez \[40\]. However, in order to keep the resulting program as close to the original as possible, we refrain from their decision to rename variables.
in the complement. Instead, the initial state is retrieved from backup variables into the original ones, just before the complement begins execution.

The success of this approach is based on a new type of compensation, according to which the final value of extracted variables is also kept in backup variables, ahead of retrieving the initial state for the complement. Accordingly, those are retrieved once the complement’s execution is over.

The proof of correctness is based on our proof method, as developed in the preceding chapter. The equivalence of the original program and its duplicated version has been proved for the extracted set of variables separately from the proof for the complementary set.

Having proved the equivalence of a program and its duplicated version, rather than expressing it as a direct transformation, the result of this chapter is also applicable for merging a duplicated statement. However, as we are interested in slice extraction for untangling code, rather than tangling it, such direction will not be further pursued (and hence the chapter’s title).

Several improvements of statement duplication will be developed in later chapters. Both the extracted code and its complement will be reduced by slicing (in the next three chapters). Then, the complement will be further reduced by reusing extracted results (Chapter 10) and redundant compensation will be eliminated in Chapter 11.

Our correctness proof has been decomposed in a certain manner such that those further improvements will be able to reuse parts of it. In particular, both Lemma 6.2 and Lemma 6.3 will be reused in Chapter 10 when reducing the complement.

Finally, we consider statement duplication as a naïve sliding operation, at least metaphorically, due to the following observation. The code of the given program statement can be printed on a single transparency slide and photocopied. Placing the two slides one on top of the other yields the original program; then sliding one away from the other and adding compensatory code would yield the duplicated version. The further improvements will refine this approach by representing a program with more slides. On each such slide, in turn, a part of the program will be printed.
Chapter 7

Semantic Slice Extraction

As was introduced earlier in the thesis (back in Chapter 1), the main challenge in slice extraction is to be able to untangle the extracted code from its complement, whilst minimizing code duplication.

With respect to the goal of minimizing duplication, it may seem self defeating to base our novel approach on statement duplication. However, this duplication can be justified by the following observation.

Once a statement has been duplicated, and each of its clones has been specialized (through copying of initial and final values, in the compensatory code) for computing only a subset of its results, we have potentially rendered some of its internal statements dead. Those can subsequently be removed by slicing.

In this chapter, requirements of slicing are derived from the earlier statement-duplication formalisation and a form of live variables analysis, to be introduced in the chapter. The result of this derivation, besides slicing requirements, is a refinement relation similar to but more general than the program equivalence of statement duplication. This refinement rule will later (in Chapter 9) be applied in deriving our first slice-extraction transformation.
7.1 Example

Going back to the \textit{sum} and \textit{prod} example, we now start with the following version:

\begin{verbatim}
    i, sum, prod := 0, 0, 1
    ; while i < a.length do
    i, sum, prod :=
        i+1, sum + a[i], prod * a[i]
    od
\end{verbatim}

and try to extract \textit{sum}. Transforming the code according to the statement-duplication program equivalence \cite{6.1} would yield the following version (replacing liveness information with local variables):

\begin{verbatim}
|[var isum, iprod, ii, fsum
; isum, iprod, ii := sum, prod, i
; i, sum, prod := 0, 0, 1
; while i < a.length do
    i, sum, prod :=
        i+1, sum + a[i], prod * a[i]
od

; fsum := sum
]| sum, prod, i := isum, iprod, ii
; i, sum := 0, 0
; while i < a.length do
    i, sum :=
        i+1, sum + a[i]
od

; fsum := sum
|]
\end{verbatim}

The duplicated statement can now be simplified by slicing. The result

\begin{verbatim}
|[var isum, iprod, ii, fsum
; isum, iprod, ii := sum, prod, i
; i, sum := 0, 0
; while i < a.length do
    i, sum :=
        i+1, sum + a[i]
od

; fsum := sum
]| sum, prod, i := isum, iprod, ii
; i, prod := 0, 1
; while i < a.length do
    i, prod :=
        i+1, prod * a[i]
od

; sum := fsum
|]
\end{verbatim}

can then be re-formatted as
7.2 Live variables analysis

Suppose we wish to perform liveness analysis on a core statement $S$ with respect to a given set of variables, $V$. Let $coV := def.S \setminus V$, the complementary set of defined variables, be considered live throughout $S$ (i.e. on exit from any of its slips). We first note that the over-approximation of considering elements of $coV$ live, even in places where they are not, will not be harmful.

Moreover, variables outside $(V, coV)$ are not defined in any slip $T$ of $S$ (since $S$ is a core statement, with no local variables). Such variables will keep their initial value throughout $S$, and hence there will be no harm in ignoring them.

Accordingly, liveness analysis in $S$ begins with $S = S[\text{live } V, coV]$ for any given $S, V$ with $coV := def.S \setminus V$. This step is correct due to Law 19 (with $V := (V, coV)$).

Then, liveness information is propagated to all slips of $S$, in a syntax-directed manner, as follows.

For sequential composition, we turn any “$(S1; S2)[\text{live } V1, coV]$” with $V1 \subseteq V$ into “$(S1[\text{live } V2, coV]; S2[\text{live } V1, coV])[\text{live } V1, coV]$”, where $V2 := (V1 \setminus ddef.S2) \cup (V \cap \text{input}.S2)$, which is correct due to Law 20 and our earlier comments on redundancy of variables outside $(V, coV)$ and the legitimacy of over-approximation (in $coV$).
For IF statements, we simply turn any “if B then S1 else S2 fi” into “if B then S1[live V1, coV] else S2[live V1, coV] fi” by applying Law 21 with V := (V1, coV).

Finally, for DO loops, we turn “(while B do S od)[live V1, coV]’” into “(while B do S[live V2, coV] od)[live V1, coV]”, where V2 := V1 ∪ (V ∩ (glob.B ∪ input.S)), which is correct due to Law 22 and, again, the earlier comment on redundancy of variables outside (V, coV).

We refer to results of liveness analysis of statement S on variables V, by saying “let T[live V1, coV] be any slip of S[live V, coV]”. In a full live variables analysis of a statement S, the set of variables, V, is not explicitly selected and the set def.S is taken in its place.

Hence, in full liveness analysis, the complementary set coV is empty and we say “let T[live V1] be any slip of S”. Then T[live V1] can be safely augmented with following assignments to any of the variables in def.S \ V1. This augmentation will still keep all auxiliary liveness information redundant (and hence removable). That is, any augmentation by assignment to dead variables (from def.S) is correct, in the context of S. (Augmentation by assignment to any other dead variable not in def.S will be correct in S[live V] but not in S itself. However, we will not be interested in such augmentations.)

Note that in deviation from traditional liveness analysis (as in [47]), typically propagating information on a flow graph, until a fixed point is reached, our algorithm requires one pass of the program’s tree. This is possible due to the simplicity of our language (e.g. no jumps) and the availability of summary information (i.e. sets def, ddef and input).

In that light, it is important and interesting to verify that our algorithm is insensitive to different parses of a given program — which are possible due to the associativity of sequential composition. That is, as much as “S1 ; (S2 ; S3)” = “((S1 ; S2) ; S3)” in our language, so will the analysis produce identical results in both cases, as is shown in the following.

**Theorem 7.1.** Distribution of liveness information over sequential composition is associative.

**Proof.** Suppose we perform liveness analysis on a core statement S with def.S = V, and we reach a slip of the form “S1 ; S2 ; S3 ” with live-on-exit variables V3. (Note that the liveness analysis guarantees V3 ⊆ V.) We now need to show that whatever the internal parsing, the liveness-analysis algorithm would identify the same results for slips S1, S2 and S3, on both “(S1 ; (S2 ; S3))[live V3]” and “((S1 ; S2) ; S3)[live V3]”.

For the former, we have

“(S1 ; (S2 ; S3))[live V3]”

= {liveness analysis (on V): let V1 := (V3 \ ddef.“S2 ; S3 ”) ∪ (V ∩ input.“S2 ; S3 ”)}

“(S1[live V1] ; (S2 ; S3)[live V3])[live V3]”
\[
\begin{align*}
&= \{\text{liveness analysis (on } V\text{): let } V2 := (V3 \setminus \text{def. } S3) \cup (V \cap \text{input } S3)\}
\quad (S1[\text{live } V1] ; (S2[\text{live } V2] ; S3[\text{live } V3])\}\text{[live } V3]\}

\text{, and for the latter, we have}
\end{align*}
\]

\[
\begin{align*}
&= \{\text{liveness analysis (on } V\text{), with } V2 \text{ as above}\}
\quad ((S1 ; S2)[\text{live } V3])
\quad (S3[\text{live } V3])\}\text{[live } V3]\}
\end{align*}
\]

Finally, we observe that \(V1 = V1'\), as expected, since

\[
\begin{align*}
V1 \\
&= \{\text{def. of } V1\}
\quad (V3 \setminus \text{def. } S2 ; S3) \cup (V \cap \text{input } S2) \quad (S1 \setminus \text{def. } S1 ; S2 \setminus S3) \\
&= \{\text{set theory: } V3 \subseteq V\}
\quad V \cap (V3 \setminus \text{def. } S2 ; S3) \cup (V \cap \text{input } S2) \\
&= \{\text{Lemma 7.2, see below}\}
\quad V \cap (((V3 \setminus \text{def. } S3) \cup \text{input } S3) \setminus \text{def. } S2) \cup \text{input } S2 \\
&= \{\text{set theory: again, } V3 \subseteq V\}
\quad (((V3 \setminus \text{def. } S3) \cup (V \cap \text{input } S3)) \setminus \text{def. } S2) \cup (V \cap \text{input } S2) \\
&= \{\text{def. of } V2\}
\quad (V2 \setminus \text{def. } S2) \cup (V \cap \text{input } S2) \\
&= \{\text{def. of } V1\}
\quad V1' \quad \blacksquare
\end{align*}
\]

**Lemma 7.2.** Let \(S1, S2, V\) be any two statements and set of variables, respectively; then

\[
(V \setminus \text{def. } S1 ; S2) \cup \text{input } S1 ; S2
\]

\[
= (((V \setminus \text{def. } S2) \cup \text{input } S2) \setminus \text{def. } S1) \cup \text{input } S1 \quad .
\]
Proof. We observe

\[(V \setminus \text{def} \ " S1 ; S2 ") \cup \text{input} \ " S1 ; S2 "\]

\[= \{ \text{def and input of ' ; ' } \}\]

\[(V \setminus (\text{def}.S1 \cup \text{def}.S2)) \cup (\text{input}.S1 \cup (\text{input}.S2 \setminus \text{def}.S1))\]

\[= \{ \text{set theory} \}\]

\[(V \setminus (\text{def}.S1 \cup \text{def}.S2)) \cup (\text{input}.S2 \setminus \text{def}.S1) \cup \text{input}.S1\]

\[= \{ \text{set theory} \}\]

\[((V \setminus \text{def}.S2) \setminus \text{def}.S1) \cup (\text{input}.S2 \setminus \text{def}.S1) \cup \text{input}.S1\]

\[= \{ \text{set theory} \}\]

\[((V \setminus \text{def}.S2) \cup \text{input}.S2) \setminus \text{def}.S1) \cup \text{input}.S1\]

\[\square\]

7.2.1 Simultaneous liveness

Liveness analysis will be useful beyond the elimination of dead assignments.

**Definition 7.3** (Simultaneous Liveness). When performing full live variables analysis on a given \(S[\text{live } V]\), a set of variables \(X\) is considered simultaneously-live (in \(S[\text{live } V]\)) if more than one element of \(X\) is on the live variables set of any slip \(T\) of \(S\). When no such slip exists, \(X\) is not simultaneously-live in \(S[\text{live } V]\).

The concept of simultaneous liveness will be useful mainly in the merging of live ranges [51]. A set of non-simultaneously-live variables can (under some further conditions) be merged into one variable. This will be explored, formalised and applied later in the thesis (see Section 8.6.2 and Appendix D).

This concludes our introduction to liveness analysis, which will be applied next for slice extraction and later in the thesis e.g. for reducing compensation after sliding (Chapter 11).

7.3 Formal derivation using statement duplication

**Refinement 7.4.** Let \(S, SV, ScoV, V, coV, iV, icoV, fV\) be three deterministic statements and five sets of variables, respectively; then
provided \( \text{def}.S = (V, coV) \),

\[
S[\text{live } V] \subseteq SV[\text{live } V],
\]

\[
S[\text{live } coV] \subseteq ScoV[\text{live } coV],
\]

\( \text{def}.SV \subseteq \text{def}.S \),

\( \text{def}.ScoV \subseteq \text{def}.S \) and

\((iV, icoV, fV) \circ \text{glob}.S \).

Proof.

\[
S
= \{ \text{duplicate statement (Program equivalence 6.1): } S \text{ is deterministic,} \}
\]

\[
\text{def}.S = (V, coV) \text{ and } (iV, icoV, fV) \circ \text{glob}.S \text{ (provisos)} \}
\]

\[
\text{“} (iV, icoV := V, coV ; S ; fV := V)
\]

\[
; V, coV := iV, icoV ; S ; V := fV)[\text{live } V, coV] \text{ ”}
\]

\[
\subseteq \{ \text{Refinement 7.5 with } S' := S \}
\]

\[
\text{“} (iV, icoV := V, coV ; SV ; fV := V)
\]

\[
; V, coV := iV, icoV ; ScoV ; V := fV)[\text{live } V, coV] \text{ ”}.
\]

Refinement 7.5. Let \( S, S', SV, ScoV, V, coV, iV, icoV, fV \) be four statements and five sets of variables, respectively; then

\[
\text{“} (iV, icoV := V, coV ; S ; fV := V)
\]

\[
; V, coV := iV, icoV ; S' ; V := fV)[\text{live } V, coV] \text{ ”}
\]

\[
\subseteq \{ \text{Refinement 7.5 with } S' := S \}
\]

\[
\text{“} (iV, icoV := V, coV ; SV ; fV := V)
\]

\[
; V, coV := iV, icoV ; ScoV ; V := fV)[\text{live } V, coV] \text{ ”}.
\]
provided
P1: \( \text{def.} S = (V, \text{co} V) \),
P2: \( \text{def.} S' = (V, \text{co} V) \),
P3: \( S[\text{live } V] \subseteq S'[\text{live } V] \),
P4: \( S'[\text{live co} V] \subseteq S[\text{co} V][\text{live co} V] \),
P5: \( \text{def.} SV \subseteq \text{def.} S \),
P6: \( \text{def.} S \circ \text{def.} S' \) and
P7: \( (iV, \text{ico} V, fV) \circ \text{glob.} S \).

\text{Proof.}

\[
\text{"} (iV, \text{ico} V := V, \text{co} V ; S ; fV := V"}
\]
\[
\text{; } V, \text{co} V := iV, \text{ico} V ; S' ; V := fV)[\text{live } V, \text{co} V] \quad \text{"}
\]
\[
\text{=} \{ \text{liveness analysis: } fV \circ \text{def.} S' \ (P7, \text{RE5}, P1 \text{ and } P2) \}
\]
\[
\text{"} ((iV, \text{ico} V := V, \text{co} V ; S ; fV := V ;
V, \text{co} V := iV, \text{ico} V ; S[\text{co} V][\text{live co} V])[\text{live fV, co} V] ; V := fV)[\text{live } V, \text{co} V] \quad \text{"}
\]
\[
\text{=} \{ \text{liveness removal: } \text{def.} S \circ fV \ (P1, P2, P6, P7 \text{ and RE5}) \}
\]
\[
\text{"} ((iV, \text{ico} V := V, \text{co} V ; S ; fV := V ;
V, \text{co} V := iV, \text{ico} V ; S[\text{co} V][\text{live co} V])[\text{live fV, co} V] ; V := fV)[\text{live } V, \text{co} V] \quad \text{"}
\]
\[
\text{=} \{ \text{liveness analysis: } (\text{input.} S \circ \text{def.} \ " fV := V ; V, \text{co} V := iV, \text{ico} V \ "\} \cap
\text{def.} \ " iV, \text{ico} V := V, \text{co} V ; S " \subseteq (iV, \text{ico} V) ; \text{then } (iV, \text{ico} V) \circ \text{def.} S \}
\]
\[
\text{"} (((iV, \text{ico} V := V, \text{co} V ; S[\text{live } V])[\text{live } V, iV, \text{ico} V] ; fV := V ;
V, \text{co} V := iV, \text{ico} V ; S[\text{co} V][\text{live fV, co} V] ; V := fV)[\text{live } V, \text{co} V] \quad \text{"}
\]
\[
\text{=} \{ \text{liveness removal: } \text{def.} S \circ (iV, \text{ico} V) \ (P5, P7, \text{RE5});
\text{then all potentially dead co} V \circ (V, iV, \text{ico} V) \ (P1, P7 \text{ and RE5})
\text{remain dead, since } \text{co} V \subseteq (\text{def.} \ " fV := V ; V, \text{co} V := iV, \text{ico} V \ "\}
\text{\textbackslash input.} \ " fV := V ; V, \text{co} V := iV, \text{ico} V \ "\}
\]
"(iV, icoV := V, coV ; SV ; fV := V ; V, coV := iV, icoV ; ScoV ; V := fV)[live V, coV] ".

7.4 Requirements of slicing

From the above law of refinement, we can gather conditions P3 and P5 as requirements for slicing. That is, for a given deterministic statement \( S \) and set of variables \( V \), any statement \( SV \) satisfying

(Q1:) \( S[live V] \subseteq SV[live V] \)

is (at least semantically) a correct slice of \( S \) with respect to \( V \). Furthermore, if condition

(Q2:) \( \text{def} \cdot SV \subseteq \text{def} \cdot S \)

holds too, we know \( SV \) can successfully replace the extracted \( S \) in a transformation of slice extraction of \( V \) from \( S \).

For sanity checking (of the generality of those requirements), we observe conditions P4 and P6 above, for the complement. There, requirement Q1 with \( S, V, SV := S', coV, ScoV \) holds due to P4 and Q2 with \( SV, S := ScoV, S' \) holds through P6. Thus, any slice \( ScoV \) of \( S' \) with respect to \( coV \) would make a good complementary statement in a transformation of slice extraction of \( V \) from \( S \).

For requirement Q1 above, we make one further observation. Following the definitions of liveness and of the relation of slice-refinement (as defined in Chapter 5), a semantic slice \( SV \) of \( S \) and any slice-refinement of those \( S \) and \( V \) is a semantic slice. That is, any \( SV \) satisfying \( S \subseteq SV \) satisfies Q1 and is thus a semantic slice of \( S, V \).

In general, any known refinement technique can be applied to \( S[live V] \) (rather than directly to \( S \)) in deriving a slice-refinement. However, in an attempt to constructively describe related transformations, a slicing algorithm will be formally developed later in the thesis.

7.4.1 Ward's definition of syntactic and semantic slices

Our semantic definition of a slice is akin to several definitions by Martin Ward (e.g. in 57, and most recently in “Conditioned Semantic Slicing via Abstraction and Refinement in FermaT” by Ward, Zedan and Hardcastle 59), and has indeed been inspired by those. Ward et al. base their semantic definition on a novel relation between programs, called semi-refinement, which involves the introduction of termination. That is, a program \( S' \) is a semi-refinement of another program \( S \) if they are both equivalent for input on which \( S \) is guaranteed to terminate. On other inputs, \( S' \) is free to terminate, or do any other thing. Recalling that refinement involves the introduction of either (or both) termination and determinism, we note that in our context of deterministic
programs, refinement and semi-refinement are the same.

In their formalism (also based on predicate transformers), a semantic slice of a given program $S$ on variables $X$ is any program $S'$ for which \[ S' ; \text{remove}(W \setminus X) \] is a semi-refinement of \[ S ; \text{remove}(W \setminus X) \], where $W$ is the final state space for $S$ and $S'$, and with \text{remove} restricting that state space. Note how our $S[\text{live} X]$ concisely captures their \[ S ; \text{remove}(W \setminus X) \]. In effect, their combination of state space restriction and semi-refinement is captured by our notation for live variables and normal refinement (and hence the requirement Q1 above) — in our context of deterministic programs.

A second relation between programs, that of a reduction, is introduced by Ward et al. to participate in the definition of a syntactic slice. A program reduction involves the replacement of substatements (i.e. slips in our terminology) with \text{skip} statements (or \text{exit}, which is beyond the scope of our investigation), thus maintaining the original syntactic structure. Then, any semantic slice of $S$ on $X$, $S'$, is also a syntactic slice, if it is a reduction of $S$. The next chapter will define the program entities of \textit{slides} to achieve a similar effect. This will allow a later formulation of a provably-correct syntax-preserving slicing algorithm.

### 7.5 Summary

This chapter has developed a refinement rule for slice extraction, based on statement duplication (from the preceding chapter) and a live variables analysis. Our approach to liveness analysis has been formalised in the chapter.

Liveness information is introduced into a program statement by first assuming all variables are live; then the information is propagated to all slips of the original statement; next, local transformations such as dead-assignment-elimination can be performed; finally, under some conditions, the correct local transformations are also globally correct such that all liveness information can be removed.

According to our new liveness-based refinement rule (Refinement 7.4) for slice extraction, both the extracted code and the complement are slices (of the same program, on two complementary sets of variables).

Advanced strategies for minimizing the amount of duplication in slice extraction will be explored later in the thesis, after developing a slicing algorithm (in Chapter 9 ahead); this will be a semantically correct algorithm, following the slicing requirements (Q1 and Q2) as derived in this chapter. That algorithm will allow us to constructively describe a transformation based on the semantic slice-extraction refinement laws of this chapter.

The next chapter will lay the foundations for our slicing algorithm by formalising a novel decomposition of programs into syntactic elements called \textit{slides}. 
Chapter 8

Slides: A Program Representation

8.1 Slideshow: a program execution metaphor

In this thesis, the execution of imperative procedural sequential programs is thought of as a systematic slideshow.

According to the slideshow metaphor, the executable code of each procedure is printed on a single transparent slide, one that is identifiable by the procedure’s unique signature.

For a reason that will soon become clear, we choose to think of the slides as A4 (or longer, if needs be) transparencies that can be projected using a classroom-like overhead projector. This is in contrast to traditional photography related slide projectors, where a picture is printed onto a film (which is then placed inside a cardboard or plastic shell) and whenever selected for viewing, is being mechanically slid away from a tray (stacking a normally prearranged collection of pictures), and onto the projector’s lamp.

It is the latter projection style that is responsible for the English terminology of a slide. Nevertheless, in a sliding transformation, to be introduced shortly and developed throughout the thesis, the sideways movement of slides will be of a somewhat different nature.

Why do we prefer overhead projection? One reason is that this way, while projecting (i.e. executing) a program, the presenter can use a non-permanent (i.e. erasable) pen for writing notes on the slide itself (or alternatively on a separate blank slide that is placed directly on top). This can be useful e.g. for keeping track of current values of local variables. The other reason is related to the order of presented slides. In tracking program execution, the order will not be as prearranged, static and sequential as is usually the case with photographic slideshows.

The slideshow is a demonstration of a typical Von-Neumann style of sequential program execution. The program itself is a collection of procedures storing imperative subprograms. (The model is probably extendable for concurrent and even truly parallel program execution, by having
either one presenter, simultaneously using multiple trays or projectors/screens, or maybe even a combination of many independent presenters/projectors.)

In the rest of this thesis, the idea of program execution as a slideshow will play no further part. (It was introduced here merely as an illustration aid.) Instead, the slideshow metaphor will be applied to the development and evolution of programs, or more specifically to slicing and refactoring.

8.2 Slides in refactoring: sliding

8.2.1 One slide per statement

It is in illustrating and formalising the slice-extraction refactoring that the program medium of slides will be instrumental. A plausible interpretation of our initial solution, that of statement duplication (from Chapter 6 above), goes as follows.

Suppose the code of a program statement $S$ is printed on a single transparency slide; duplicate that slide, thus yielding two clones, say $S_1$ and $S_2$; place them one on top of the other (thus getting the original $S$); slide one of them (say $S_2$) sideways; finally, for behaviour preservation, add compensatory code.

But duplication of code is bad. The interpretation of our first step for reducing such duplication, (as was defined in the preceding chapter and will be automated in the next), in terms of sliding, is described in what follows.

8.2.2 A separate slide for each variable

In a first step forward, we will no longer think of a statement $S$ as being printed on a single slide. Instead, we take further advantage of features of transparency slides, and dedicate a separate slide for each defined variable. On each such slide, the slice of that variable (from the end of $S$) can be printed.

Assuming no dead code, it can be shown that the union of such slides is $S$ itself. Then, when a set of variables $V$ is selected for extraction from $S$, all slides of variables in the complementary set $coV := def.S \setminus V$ can be separated from slides of $V$ by sliding. As in the previous solution, compensatory code should be added, to ensure behaviour preservation.

Another feature of transparency slides that proves useful here is that the relative location of slid program elements remains the same. This is a fact that existing approaches for syntax-preserving slice extraction, e.g. KH03 [39], have struggled with, both in illustration and formalisation. With the slideshow metaphor in mind, this requirement has become relatively trivial.

The result is the extraction of a slice (of $V$), with the complement being also a slice, of the
complementary set $coV$. But the complement can be made even smaller, by reusing the extracted results of $V$, as in the following.

### 8.2.3 A separate slide for each individual assignment

Instead of having a slide for each (defined) variable, our final improvement will involve designating a separate slide for each individual assignment. On each such slide we shall print the assignment itself, and all guards (controlling whether the assignment will or will not be executed). We shall pay special attention to preserving layout (on the slide, both metaphorically and later when formalising slides), such that the original program will be reproducible, as the union of all slides. Similarly, each slice will consist of the union of all slides of included assignments.

This time, when asked to extract variables $V$ from $S$, all slides in the slice of $V$ will be separated from the remaining slides by sliding, leaving a potentially smaller complement. However, for preserving behaviour, some extra measures will need to be taken. These include duplication of some slides (that must appear in both the extracted slice and its complement) and the renaming of reused extracted values in the complement.

This sliding transformation, along with the extra measures, will be formalised later in the thesis (see chapter 10). Then, the need for renaming reused extracted values, in the complement, will be removed in Chapter 11.

### 8.3 Representing non-contiguous statements

For slice extraction to be implemented as a sliding operation, we need to decompose a given statement into a set of not-necessarily contiguous statements. As was introduced earlier, in Section 4.1.1 instead of speaking of substatements as parts of a program statement, we speak of slips and slides. In terms of the abstract syntax tree, the former correspond to a subtree and the latter to a path from the root to a node. More precisely, a slide is a statement, formed by that path, replacing any statement child (i.e. slip) of a node on the path, which is itself not on the path, with the empty statement `skip`. For convenience, in concrete examples, we avoid mentioning the `skip`, leaving an empty space instead. Note that this space is empty but considered transparent, in contrast to the misleading convention of “whitespace”. (Admittedly, a concrete syntax that understands such empty spaces as the empty statement would have been preferable.)

For example, the program on the left-hand column can be represented as the union of the slides of its individual assignments.
Let $S$ be any core statement and $V$ be any set of variables; then

\[
\text{slides}.S.V \triangleq \begin{cases} \text{skip} & \text{when } V \circ \text{def}.S; \\ \text{otherwise we have the following definitions:} & \\ \text{slides}.“ X1,X2 ::= E1,E2 ”.V \triangleq “ X1 ::= E1 ” & \text{where } X1 \subseteq V \text{ and } X2 \circ V; \\ \text{slides}.“ S1 ; S2 ”.V \triangleq “ \text{slides}.S1.V) ; (\text{slides}.S2.V)” ; \\ \text{slides}.“ \text{if } B \text{ then } S1 \text{ else } S2 \text{ fi ”.V} \triangleq “ \text{if } B \text{ then } \text{slides}.S1.V \text{ else } (\text{slides}.S2.V) \text{ fi } ” ; \\ \text{slides}.“ \text{while } B \text{ do } S \text{ od ”.V} \triangleq “ \text{while } B \text{ do } (\text{slides}.S.V) \text{ od } ”. \\
\end{cases}
\]

Figure 8.1: Computing the slides of a core statement with respect to a set of variables.

\[
\begin{align*}
\text{if } x>y \text{ then} \\
\text{m} := x \\
\text{else} \\
\text{m} := y \\
\text{fi} = \\
\text{if } x>y \text{ then} \\
\text{m} := x \\
\text{else} \\
\text{fi} \\
\text{fi} \\
\end{align*}
\]

Such a union operation will be formalised shortly (in the next section) and such slides of individual assignments will be formalised later in the chapter (in Section 8.6). But first, we find it more convenient to formalise a more coarse-grained concept. We define the statement formed by the union of all individual-assignment slides of a certain program statement $S$ with respect to a set of variables $V$. (This way, we avoid having to formalise the access to an individual assignment, \textit{e.g.} through labels.)

For a given program statement $S$ and any set of variables $V$, we define the subprogram of $S$ containing all assignments to variables in $V$, along with all their enclosing compound statements, as $\text{slides}.S.V$ (see Figure 8.1). In the example above, say the statement on the left is $S$, then the statement $\text{slides}.S\{x,y,z\}$ is the empty statement $\text{skip}$, whereas $\text{slides}.S\{m\}$ is the union of the two individual slides (which is in this case the whole program, $S$).

Note that in general, a given core statement $S$ is represented as $\text{slides}.S.\{\text{def}.S\}$. Further note that we choose to name that function $\text{slides}$, instead of say $\text{slide}$ or $\text{slideFor}$, despite the fact that it yields a single statement, because its result should be thought of as the collection of all individual-assignment slides of variables in the selected set. For convenience, we choose not to distinguish that collection from the actual statement its union would yield. Indeed, when collecting a set of slides, putting them one on top of the other, the resulting program is the union of those slides. This is formalised next.
Let $S_1, S_2, B, X_1, X_2, X_3, E_1, E_2, E_3$ be two core statements, a boolean expression, three sets of variables and three corresponding expressions, respectively; then

$$S_1 \cup \text{skip} \triangleq S_1;$$
$$\text{skip} \cup S_2 \triangleq S_2;$$

" $X_1, X_2 := E_1, E_2 $ " $\cup " X_1, X_2, X_3 := E_1, E_2, E_3 "$ provided $X_2 \diamond X_3; $

" if $B$ then $S_1$ else $S_2$ fi $\cup " \text{if } B \text{ then } S_1' \text{ else } S_2' \text{ fi } "$ $\triangleq$

" while $B$ do $S_1$ od " $\cup " \text{while } B \text{ do } S_1' \text{ od } "$ $\triangleq "$.

Figure 8.2: Unifying (or merging) statements.

### 8.4 Collecting slides: the union of non-contiguous code

We define an operation for unifying (or merging) two program statements, $S_1$ and $S_2$, into a single statement, $S_1 \cup S_2$ (see Figure 8.2).

Note that two statements $S_1$ and $S_2$ are unifiable (i.e. $S_1 \cup S_2$ is well-defined) only when they have the same shape, as is implicitly expressed in the definition of $\cup$. For example, an IF statement can only be merged with an empty statement or with another IF statement whose guard and two branches are unifiable with the corresponding guard and branches of the former.

Furthermore, note that we do not write "$ S_1 \cup S_2 "$ and do not define wp-semantics for slide union (or for taking slides in general). This is so since $\cup$ is not a construct of our programming language. It is rather a meta-program operation, generating (when well-defined) a program statement.

Following its definition, it is easy to verify that the union of statements is commutative, associative and idempotent; hence the choice of infix $\cup$. The following theorem shows that the union of slides (for a given statement and a pair of variable sets) is equivalent to the slides of the union.

**Theorem 8.1.** Any pair of slides of a single statement, $\text{slides}.S.V1$ and $\text{slides}.S.V2$, is unifiable. Furthermore, we have

$$(\text{slides}.S.(V_1 \cup V_2)) = ((\text{slides}.S.V1) \cup (\text{slides}.S.V2)) .$$

The proof, by induction on the structure of $S$, can be found in Appendix C.
8.5 Slide dependence and independence

Data flow between sets of slides is formalised through a relation of slide dependence and a corresponding concept of slide independence. We start with the latter.

**Definition 8.2** (Slide Independence). A set of variables $V$ is considered *slide independent* with respect to a given statement $S$, if the condition

$$\text{glob.(slides.$S$.V) \cap \text{def.$S$} \subseteq V}$$

holds. (Recall $\text{slides.$S$.V}$ is a normal statement, so $\text{glob.(slides.$S$.V)}$ is the set of global variables in that statement.)

Interesting (semantic) properties of independent slides will be extensively investigated in the next chapter, when developing a slicing algorithm. The complementary notion, of *slide dependence*, is defined as follows.

**Definition 8.3** (A Relation of Slide Dependence). A set of variables $V_1$ depends on another set $V_2$ with respect to a given statement $S$, i.e. $V_1$ is related to $V_2$ through slide dependence, when

$$\text{input.(slides.$S$.V1) \cap \text{def.(slides.$S$.V2)} \neq \emptyset}$$

8.5.1 Smallest enclosing slide-independent set

The reflexive transitive closure of a set $V$ in the context of slides.$S$, denoted $V^*$, is the smallest slide-independent superset of $V$. (Recall that slide dependence is indeed a relation between sets of variables.)

When asked to compute the reflexive transitive closure of slide dependence, for a given statement $S$ and set of variables $V$, we choose to avoid computing the full relation. Instead, we take a faster lazy approach, repeatedly adding (to $V$) slides on which $V$ depends, until a fixed point is reached. At each step, all variables in $U := \text{input.(slides.$S$.V) \cap \text{def.$S$}$, are added to $V$. A fixed point is reached when $U \subseteq V$.

This can be slightly improved by observing the relationship between global variables and input. In general, we note that computing the set of global variables, in a collection of slides (as for any statement), is faster than computing its set of input variables. Now from RE5 we
know \( \text{glob}.T = \text{def}.T \cup \text{input}.T \). So, for computing the set \( U \), we observe that even though \( \text{glob}.(\text{slides}.S.V) \cap \text{def}.S \) may be larger than \( \text{input}.(\text{slides}.S.V) \cap \text{def}.S \), the extra variables would be from \( \text{def}.(\text{slides}.S.V) \) and hence from \( V \). We conclude that since the only purpose of \( U \), in the present algorithm, is to be tested for inclusion in \( V \) (and then possibly be added to \( V \)), there would be no harm in including variables from \( V \) in \( U \).

Thus the algorithm for computing the reflexive transitive closure of slide dependence, for any given \( S \), \( V \) is as follows:

\[
\text{slides-dep-rtc}.S.V \triangleq \begin{cases} \text{if } U \subseteq V \text{ then } V \text{ else } \text{slides-dep-rtc}.S.(V \cup U) \end{cases}
\]

where \( U := \text{glob}.(\text{slides}.S.V) \cap \text{def}.S \).

8.6 SSA form

Up till now we had one slide for each variable, including all definitions of that variable. Can we refine the representation such that each slide will be dedicated to a specific instance (i.e. definition point) of a variable?

We do that by splitting the selected variable, such that a new variable is defined at each definition point, in the style of SSA. We then show that under some conditions the instances of a variable can be merged back (to the original) even after performing some transformations (e.g. slicing).

8.6.1 Transform to SSA

The set of instance variables in the SSA form replacing a variable of the original program are expected to maintain a property of no-simultaneous-liveness. This way, it will be possible to transform the program back from SSA.

A to\( \text{SSA} \) algorithm is formally derived for our core language as Transformation D.5 in Appendix D and is repeated here as Figure 8.3.

In transforming a given statement \( S \) with respect to variables \( X \) (i.e. splitting definitions of \( X \) alone), we aim to end up with a statement \( S' \) free of occurrences of \( X \) (i.e. \( \text{glob}.S' \cap X \)) and with at most one instance (of each member of \( X \)) live at each point of \( S' \).

According to Transformation D.5 and its corresponding preconditions P1-P7, we observe that \( X \) should be partitioned into six mutually-disjoint subsets \( X_1, X_2, X_3, X_4, X_5, X_6 \). However, following postcondition Q1 and preconditions P5 and P6, we further observe that of those, only \( X_4 \) and \( X_5 \) are both live-on-exit and defined in \( S \). Since in general we mean to transform all variables in \( X := \text{def}.S \), and since we expect all members of \( X \) to be live-on-exit, we are left with
Let $S, X, Y$ be any core statement and two (disjoint) sets of variables; let $X_1, X_2, X_3, X_4, X_5$ be five (mutually disjoint) subsets of $X$, and let $X_{L1i}, X_{L2i}, X_{L3i}, X_{L4i}, X_{L5f}$ be six sets of instances, all included in the full set of instances $XL$s; let $S'$ be the SSA form of $S$ defined by

$$S' := \text{toSSA}((S, X, (X_{L1i}, X_{L2i}, X_{L3i}, X_{L4i}), (X_{L3i}, X_{L4f}, X_{L5f}), Y, XLs);$$

then (Q1:)

" $\text{(}S; X_{L3i}, X_{L4f}, X_{L5f} := X_3, X_4, X_5)[\text{live } X_{L3i}, X_{L4f}, X_{L5f}, Y] \text{ ) } =$

" $\text{(}X_{L1i}, X_{L2i}, X_{L3i}, X_{L4i} := X_1, X_2, X_3, X_4; S')[\text{live } X_{L3i}, X_{L4f}, X_{L5f}, Y] \text{ ) }$

and (Q2: $X \circ \text{glob.S'}$

provided

P1: $\text{glob.S} \subseteq (X, Y)$,

P2: $(X_1, X_2, X_3, X_4, X_5) \subseteq X$,

P3: $(X_{L1i}, X_{L2i}, X_{L3i}, X_{L4i}, X_{L4f}, X_{L5f}) \subseteq XLs$,

P4: $XLs \circ (X, Y)$,

P5: $(X_1, X_3) \circ \text{def.S}$,

P6: $(X_2, X_4, X_5) \subseteq \text{def.S}$ and

P7: $(X \cap ((X_3, X_4, X_5) \setminus \text{ddef.S} \cup \text{input.S})) \subseteq (X_1, X_2, X_3, X_4)$ .

Figure 8.3: Transformation D.5 of toSSA (from the appendix).
the 2-partition $X = (X_4, X_5)$. Of those, we observe (by P2,P7) that only members of $X_4$ are
live-on-entry (i.e. in $(X \setminus \operatorname{def}.S) \cup (X \cap \operatorname{input}.S)$).

We now need to prepare initial and final instances for $X_4$ and $X_5$. For the former, we observe
how Q1 and P3 imply that the set of initial instances $XLAi$ must be disjoint from the set $XLAf$ of
final instances, whereas for the latter only final instances ($XLf$) are required. From P2,P3 and
P4 it is clear that all instances must be fresh (i.e. disjoint from $(X, Y)$).

Thus, the $toSSA$ algorithm can be applied as follows:
Let $S$ be any given statement; let $X := \operatorname{def}.S$ and $Y := (\operatorname{glob}.S \setminus X)$ be a 2-partition of
$\operatorname{glob}.S$; let $X_4 := (X \setminus \operatorname{def}.S) \cup (X \cap \operatorname{input}.S)$ and $X_5 := X \setminus X_4$ be a 2-partition of $X$;
let $(XLAi, XLAf, XLf) := \text{fresh.}((X_4, X_4, X_5), (X, Y))$ be three sets of fresh instances — ac-
ccording to property Q2 of our definition of fresh (see Section 3.1.3) we indeed get the required
$(XLAi, XLAf, XLf) \circ (X, Y)$ — and finally let
$S' := toSSA(S, (X_4, X_5), XLAi, (XLAf, XLf), Y, (XLf))$ and let $XLim := \operatorname{glob}.S' \setminus
(Y, XLAi, XLAf, XLf)$ be the set of all intermediate instances; we then observe

$S$

= \{\text{intro aux. liveness info.; intro. dead assignment;}

\text{intro. self-assignment; assignment-based sub.}\}

= \{\text{prop. liveness info.}\}

\begin{align*}
\langle S ; XLAf, XLf & := X_4, X_5 ; X_4, X_5 := XLAf, XLf][\text{live } X_4, X_5] \rangle \\
& = \{\text{Q1 of Transformation D.5 P1-P7 hold by construction (as justified above)}\}
\end{align*}

\begin{align*}
\langle (XLAi := X_4 ; S')[\text{live } XLAf, XLf] ; X_4, X_5 := XLAf, XLf][\text{live } X_4, X_5] \rangle \\
& = \{\text{def. of live: def. of $XLim$; note $XLim \circ (X_4, X_5)$ due to}
\text{Q2 of toSSA ($X \circ \operatorname{glob}.S'$)}\}
\end{align*}

\begin{align*}
\langle \text{[var } XLAi, XLAf, XLf, XLim ; XLAi := X_4 ; S'; X_4, X_5 := XLAf, XLf]| \rangle \\
& = .
\end{align*}

8.6.2 Back from SSA

As the derivation above is made of program equations, the reversed derivation is used for returning
from SSA. However, returning from SSA is more general since we wish to return even after having
made some transformations on the immediate SSA version. In effect, returning from SSA involves
Let $S'$ be any core statement and $(XL_1 \cup XL_2) \subseteq XL_s$; let $S$ be a statement defined by $S := \text{merge-vars}(S', XL_s, X, XL_1, XL_2, Y)$; then (Q1:)

\[
(XL_1 := X_1 \; ; \; S')[\text{live } XL_2, Y] = (S \; ; \; XL_2 := X_2)[\text{live } XL_2, Y]
\]

and (Q2:) $XLs \circ \text{glob}.S$

provided

P1: $\text{glob}.S' \subseteq (XL_s, Y)$,
P2: $(XL_1 \cup XL_2) \subseteq XL_s$,
P3: $(X_1 \cup X_2) \subseteq X$,
P4: $X \circ (XL_s, Y)$,
P5: no two instances of any member of $X$ are sim.-live at any point in $S'[\text{live } XL_2, Y]$,
P6: $(XL_s \cap ((XL_2 \setminus \text{def}.S') \cup \text{input}.S')) \subseteq XL_1$,
P7: no def-on-live: i.e. no instance is defined where another instance is live-on-exit,
P8: no multiple-defs: i.e. each assinment defines at most one instance (of any $X_i$).

Figure 8.4: Transformation [D.6] of merge-vars (from the appendix).

the merge of all instances of an original program variable. This way, all definitions of pseudo instances become redundant self assignments and hence removed.

Accordingly, the fromSSA algorithm is defined to call the more general merge-vars algorithm, as derived in Transformation [D.6] in Appendix [D] and is repeated here as Figure [8.4] Thus, fromSSA is defined as

\[
\text{fromSSA}.(S', X, XL_1, XL_f, Y, XLs) \triangleq \text{merge-vars}(S', XL_s, X, XL_1, XL_f, Y)
\]

where $S'$ is a statement in SSA form with respect to variables $X$, variables $X_1$ are live-on-entry with corresponding initial instances $XL_1$, $XL_f$ are final instances of $X$, variables $Y$ are non-SSA program variables, and $XL_s$ is the complete set of instances of $X$.

In the following, we show that the toSSA algorithm is invertible, with fromSSA its inverse.

### 8.6.3 SSA is de-SSA-able

When a statement in SSA can be converted back, thus merging all instances of each transformed variable, to its original name, we say it is de-SSA-able. In the following theorem we prove that toSSA yields a de-SSA-able statement. This result is not surprising. It was expected and is stated here for ‘sanity checking’. However, the theorem will actually become useful a little later, when
proving de-SSA-ability of SSA-based slices. There, we shall combine the result of this theorem
with an observation that slicing preserves de-SSA-ability.

**Theorem 8.4.** Let \( S \) be any core statement and let \( X, Y := (\text{def}.S), (\text{glob}.S \setminus \text{def}.S), X1 := (X \cap ((X \setminus \text{ddef}.S) \cup \text{input}.S) \) and \((XLi, XLf) := \text{fresh}.((X1, X), (X, Y))\); let \( S' \) be the SSA version of \( S \), defined as \( S' := \text{toSSA}.(S, X, XLi, XLf, Y, (XLi, XLf)); \) then \( S' \) is de-SSA-able. That is, all preconditions, P1-P8, of the fromSSA algorithm hold for
\( S'' := \text{fromSSA}(S', XLS, X, XLi, XLf, Y) \) where \( XLS := ((XLi, XLf) \cup (\text{def}.S1' \setminus Y)). \)

The proof can be found in appendix D.

### 8.7 Summary

A program representation of non-contiguous statements has been defined along with an operation
for merging such statements. Slides — following an original program execution metaphor that has
been introduced — encompass control dependences. In the context of our simply structured lan-
guage, this is done in a syntax-directed manner. The complementary notion of data dependences
has been captured by a relation of slide dependence. Together, those will take part in computing
slices, in the next chapter.

For any given statement \( S \), the function \( \text{slides}.S \) takes a set of variables, say \( V \), and yields a
statement \( (\text{slides}.S, V) \) which includes the union of individual-assignment slides of all assignments
to variables in \( V \). That way, we have avoided the need to formalise labels of internal program
points (for distinguishing one assignment of a variable from another).

Instead, the finer-grained level of individual-assignment slides has been made accessible through
the development of transformations to and from the popular static single assignment (SSA) form.
Slides of a particular instance of a variable, on the SSA form, correspond to the individual assign-
ment of that instance, on the original program. The SSA form and its related slides will help in
the next chapter to turn a naive flow-insensitive slicing algorithm into a flow-sensitive one.
Chapter 9

A Slicing Algorithm

This chapter develops a provably correct slicing algorithm. The algorithm is based on the observation that a slide-independent collection of slides yields a semantically correct slice.

The algorithm’s development will consist of two stages. A first attempt will produce crude (i.e., too large) slices. Then, by adopting the refined program representation of SSA-based slides, the same algorithm will be shown to produce refined (i.e., smaller, more accurate and desirable) slices.

9.1 Flow-insensitive slicing

The observation that independent slides yield correct slices is proved in the following.

**Theorem 9.1.** Let $S$, $V$ be a core statement and set of variables, respectively. Then provided $V$ is slide independent in $S$ (i.e., $\text{glob.}(\text{slides}.S.V) \cap \text{def}.S \subseteq V$), $\text{slides}.S.V$ is a slice-refinement of $S$ with respect to $V$.

**Proof.** The proof is by induction on the structure of $S$. We assume that for any slip $T$ of $S$ (for which $\text{slides}.T.V$ is independent in $T$, as is guaranteed by Lemma [9.2]), we have $[\text{wp}.T.Q \Rightarrow \text{wp}.(\text{slides}.T.V).Q]$ for all $Q$ with $\text{glob}.Q \cap \text{def}.S \subseteq V$. We then prove that provided $V$ is slide independent in $S$, we have $[\text{wp}.S.P \Rightarrow \text{wp}.(\text{slides}.S.V).P]$ for all such $P$ with $\text{glob}.P \cap \text{def}.S \subseteq V$.

First, if $V \circ \text{def}.S$ we observe for all $P$ with $\text{glob}.P \cap \text{def}.S \subseteq V$ (i.e., $\text{glob}.P \circ \text{def}.S$):

\[
\begin{align*}
\text{wp}.(\text{slides}.S.V).P &= \{ \text{slides when } V \circ \text{def}.S \} \\
\text{wp}.\text{skip}.P &= \{ \text{wp of skip} \} \\
\end{align*}
\]

\[P\]
\[ \Leftarrow \{ \text{pred. calc.} \} \]
\[ P \land \text{wp} \cdot S \cdot \text{true} \]
\[ = \{ \text{RE3: } \text{glob} \cdot P \circ \text{def} \cdot S \} \]
\[ \text{wp} \cdot S \cdot P \]

In the remaining cases we shall assume \( V \cap \text{def} \cdot S \neq \emptyset \).

\( S = " \ X := E \ " : \) We observe for all \( P \) with \( \text{glob} \cdot P \cap \text{def} \cdot S \subseteq V \)
\[ \text{wp} \cdot " \ X := E \ " \cdot P \]
\[ = \{ \text{wp of } \ ::= \}' \} \]
\[ P[X \setminus E] \]
\[ = \{ \text{remove redundant subs.: let } X_1 := X \cap V \text{ and the proviso ensures} \]
\[ \text{glob} \cdot P \cap X \subseteq X_1 \}
\[ P[X_1 \setminus E_1] \]
\[ = \{ \text{wp of } \ ::= \}' \} \]
\[ \text{wp} \cdot " \ X := E_1 \ " \cdot P \]
\[ = \{ \text{slides of } \\ ::= \}' \} \]
\[ \text{wp} \cdot (\text{slides} \cdot " \ X := E \ " \cdot V) \cdot P \]

\( S = " \ S_1 ; \ S_2 \ " : \) We observe for all \( P \) with \( \text{glob} \cdot P \cap \text{def} \cdot S \subseteq V \)
\[ \text{wp} \cdot " \ S_1 ; \ S_2 \ " \cdot P \]
\[ = \{ \text{wp of } \ ; \ \} \]
\[ \text{wp} \cdot S_1 \cdot (\text{wp} \cdot S_2 \cdot P) \]
\[ \Rightarrow \{ \text{ind. hypo.: } \text{glob} \cdot P \cap \text{def} \cdot S \subseteq V \text{ and } V \text{ is slide ind. in } S_2 \} \]
\[ \text{wp} \cdot S_1 \cdot (\text{wp} \cdot (\text{slides} \cdot S_2 \cdot V) \cdot P) \]
\[ \Rightarrow \{ \text{ind. hypo.: } \text{glob} \cdot (\text{wp} \cdot (\text{slides} \cdot S_2 \cdot V) \cdot P) \cap \text{def} \cdot S \subseteq V \text{ due to RE2} \]
\[ \text{since both } \text{glob} \cdot P \cap \text{def} \cdot S \subseteq V \text{ and} \]
\[ \text{input} \cdot (\text{slides} \cdot S_2 \cdot V) \cap \text{def} \cdot S \subseteq V \text{ (slide ind. of } V \text{ in } S_2); \]
\[ V \text{ is slide ind. in } S_1 \}
\[ \text{wp} \cdot (\text{slides} \cdot S_1 \cdot V) \cdot (\text{wp} \cdot (\text{slides} \cdot S_2 \cdot V) \cdot P) \]
\[ = \{ \text{wp of } \ ; \ \} \]
\[ \text{wp} \cdot " \ (\text{slides} \cdot S_1 \cdot V) ; \ (\text{slides} \cdot S_2 \cdot V) \ " \cdot P \]
\[ = \{ \text{slides of } \ ; \ \} \]
\[ \text{wp} \cdot (\text{slides} \cdot " \ S_1 ; \ S_2 \ " \cdot V) \cdot P \]
$S =$ "if $B$ then $S_1$ else $S_2$ fi": We observe for all $P$ with $\text{glob}.P \cap \text{def}.S \subseteq V$

wp." if $B$ then $S_1$ else $S_2$ fi".P = 

\{ wp of IF \}

$(B \Rightarrow \text{wp}.S_1.P) \land (\neg B \Rightarrow \text{wp}.S_2.P)$

\Rightarrow \{ ind. hypo., twice: $\text{glob}.P \cap \text{def}.S \subseteq V$ and $V$ is slide ind. in both $S_1$ and $S_2$ \}

$(B \Rightarrow \text{wp}.(\text{slides}.S_1.V).P) \land (\neg B \Rightarrow \text{wp}.(\text{slides}.S_2.V).P)$

= \{ wp of IF \}

\text{wp." if } B \text{ then } \text{slides}.S_1.V \text{ else } \text{slides}.S_2.V \text{ fi".P}

= \{ \text{slides of IF} \}

\text{wp.(slides." if } B \text{ then } S_1 \text{ else } S_2 \text{ fi".V).P} .

$S =$ "while $B$ do $S_1$ od": We observe for all $P$ with $\text{glob}.P \cap \text{def}.S \subseteq V$

wp." while $B$ do $S_1$ od".P = 

\{ wp of DO: $[k.Q \equiv (B \vee P) \land (\neg B \lor \text{wp}.S_1.Q)] \}$

$(\exists i : 0 \leq i : k^i.\text{false})$

\Rightarrow \{ \text{see below: } [l.Q \equiv (B \vee P) \land (\neg B \lor \text{wp}.(\text{slides}.S_1.V).Q)] \}$

$(\exists i : 0 \leq i : l^i.\text{false})$

= \{ wp of DO with $l$ as above \}

\text{wp." while } B \text{ do (slides}.S_1.V \text{) od".P}

= \{ \text{slides of DO} \}

\text{wp.(slides." while } B \text{ do } S_1 \text{ od".V).P} .

We finish by proving for the second step above, by induction, having $[k^i.\text{false} \Rightarrow l^i.\text{false}]$ for all $i$, provided $[\text{wp}.S_1.P \Rightarrow \text{wp}.(\text{slides}.S_1.V).P]$ for all $P$ with $\text{glob}.P \cap \text{def}.S \subseteq V$ (induction hypothesis above).

The base case ($i = 0$) is trivial ($[\text{false} \Rightarrow \text{false}]$, recall the definition of function iteration). Then, for the induction step, we assume $[k^i.\text{false} \Rightarrow l^i.\text{false}]$ and prove $[k^{i+1}.\text{false} \Rightarrow l^{i+1}.\text{false}]$.

$k^{i+1}.\text{false} = \{ \text{def. of func. it.} \}

k.(k^i.\text{false}) = \{ \text{def. of } k \}

(B \lor P) \land (\neg B \lor \text{wp}.S_1.(k^i.\text{false}))
⇒ \{\text{ind. hypo.}\}  \\
(B \lor P) \land \neg (B \lor \text{wp}.S1.(l^i.false))  \\
⇒ \{\text{slide ind., proviso and glob.}(l^i.false) \cap \text{def}.S \subseteq V\text{ since}\}  \\
\quad ((\text{glob}.B \cup \text{glob}.P \cup \text{input}.S1) \cap \text{def}.S) \subseteq V\}  \\
(B \lor P) \land \neg (B \lor \text{wp}.(\text{slides}.S1.V).(l^i.false))  \\
= \{\text{def. of } l\}  \\
\quad l.(l^i.false)  \\
= \{\text{def. of func. it.}\}  \\
\quad l^{i+1}.false  \\
\square

**Lemma 9.2.** Let $S$ be any core statement (i.e. no local variable scopes); let $V$ be a set of slide-independent variables (in $S$); let $T$ be any slip of $S$; then $V$ is also slide independent in $T$. That is,

$$\text{glob.}(\text{slides}.T.V) \cap \text{def}.T \subseteq V.$$  

**Proof.**

$$\begin{align*}
glob.(\text{slides}.T.V) \cap \text{def}.T  \\
\subseteq & \quad \{\text{Lemma 9.3}\}  \\
glob.(\text{slides}.S.V) \cap \text{def}.T  \\
\subseteq & \quad \{\text{Lemma 9.4}\}  \\
glob.(\text{slides}.S.V) \cap \text{def}.S  \\
\subseteq & \quad \{\text{proviso (V is slide ind. in S)}\}  \\
& \quad V.  \\
\square
\end{align*}$$

The proofs of the remaining lemmata are given in Appendix C.

**Lemma 9.3.** Let $S$ be any core statement; let $T$ be any slip of $S$ and let $V$ be any set of variables; then

$$\text{glob.}(\text{slides}.T.V) \subseteq \text{glob.}(\text{slides}.S.V)$$  

**Lemma 9.4.** Let $S$ be a core statement; let $T$ be any slip of $S$; then

$$\text{def}.T \subseteq \text{def}.S.$$
Given a core statement $S$ and variables of interest $V$, compute the flow-insensitive slice, \( \text{fi-slice}.S.V \), as follows:

\[
\text{fi-slice}.S.V \triangleq \text{slides}.S.V^*
\]

where $V^* := \text{slides-dep-rtc}.S.V$ with \( \text{slides-dep-rtc}.S.V \) defined recursively as follows:

\[
\text{slides-dep-rtc}.S.V \triangleq \text{if } U \subseteq V \text{ then } V \text{ else slides-dep-rtc}.(V \cup U) \text{ fi}
\]

where $U := \text{glob}.(\text{slides}.S.V) \cap \text{def}.S$.

Figure 9.1: A flow-insensitive slicing algorithm.

9.1.1 The algorithm

Our flow-insensitive slicing algorithm is given in Figure 9.1. Given a core statement $S$ and a set of variables $V$, the algorithm first computes the smallest possible slide-independent superset $V^*$ of $V$ (i.e. the reflexive transitive closure of the slide dependence of $S$ on $V$, as in Section 8.5.1); then, the union of slides of $S$ on $V^*$, i.e. the statement $\text{slides}.S.V^*$, is produced as the slice of $S$ on $V$.

The algorithm is correct in the sense that, for any $S$ and $V$, we get \( \text{fi-slice}.S.V \) as a valid slice of $S$ with respect to $V$. That is, requirements Q1 and Q2 of slicing both hold.

For the former ($S[\text{live } V] \subseteq (\text{fi-slice}.S.V)[\text{live } V]$), suffice it to show that $S \subseteq V \text{ fi-slice}.S.V$ holds. This is so due to Theorem 9.1 with $V := V^*$ (which gives $S \subseteq V^* \text{ fi-slice}.S.V$) and since $V \subseteq V^*$ (by construction of $\text{slides-dep-rtc}.S.V$). To see why this results in $S \subseteq V \text{ fi-slice}.S.V$, recall the definition of slice-refinement, from which we indeed get for all predicates $P$ with $\text{glob}.P \subseteq V \subseteq V^*$ the required $[\text{wp}.S.P \Rightarrow \text{wp}.(\text{fi-slice}.S.V).P]$.

The latter ($\text{def}.(\text{fi-slice}.S.V) \subseteq \text{def}.S$) holds since all defined variables in any set of slides of $S$ are also defined in $S$ itself, as the following lemma (which is proved in Appendix C) shows.

**Lemma 9.5.** Let $S, V$ be any core statement and set of variables, respectively; then

\[
\text{def}.(\text{slides}.S.V) \subseteq \text{def}.S
\]

9.1.2 Example

The crudeness of this flow-insensitive slicing is demonstrated with the following example. Slicing for \textit{sum} on the program on the left-hand column would yield the program on the right.
The second loop is unnecessarily included since the slide of \( \text{sum} \) depends on the slide of \( i \), which in turn includes all assignments to \( i \) (along with their enclosing control structures). Nevertheless, this result is still a correct slice, with respect to \( \text{sum} \). But we can do better, as is shown next.

### 9.2 Make it flow-sensitive using SSA-based slides

Now that we have a provably correct slicing algorithm, we refine it by making it sensitive to control flow. This will lead to (potentially) smaller, more accurate slices. (As a bonus, the refined algorithm will be applicable for the more traditional backward slicing, in which slicing criteria may refer to internal program points. However, this kind of slicing will not be pursued.)

The key to turning the flow-insensitive slicing algorithm into flow-sensitive, lies in splitting of variables. The previous algorithm, in an attempt to stay on the safe side, added the slides of all assignments to a variable as soon as any of those slides was needed — completely ignoring the program point(s) in which the value of that variable was used and the order of execution of substatements.

By splitting a variable into several instances, such that each definition point introduces a new variable, as in our SSA-like form (see Section 8.6 of the preceding chapter), the existing algorithm can gain flow-sensitivity.

#### 9.2.1 Formal derivation of flow-sensitive slicing

Take a core statement \( S \) and any set of variables of interest \( V \). Transform \( S \) to SSA form and take the flow-insensitive slice of the final instances of \( V \). Finally, all instances can be merged (back to the original name).

\[
\text{“ } S[\text{live } V] \text{ “}
\]
\[\begin{align*}
= \{Q1 \text{ of Transformation } D.5\text{ with} \\
\quad S' := toSSA.(S, (V, coV), (VL1i, coVL1i), (VLf, coVLf), \\
\quad ND, (VL1i, coVL1i, VLf, coVLf)), \text{ def. } S = (V, coV), \\
\quad ND := \text{glob. } S \setminus (V, coV), V1, coV1 := (V \cap \text{input. } S), (coV \cap \text{input. } S) \text{ and} \\
\quad (VL1i, coVL1i, VLf, coVLf) := \text{fresh.}((V1, coV1, V, coV), \text{glob.} )) \\
\quad (VL1i, coVLi := V1, coV1 ; S' ; V, coV := VLf, coVLf)][\text{live } V] \\
= \{\text{remove dead assignment (Law 23): } coV \circ V\} \\
\quad (VL1i, coVL1i := V1, coV1 ; S' ; V := VLf)][\text{live } V] \\
= \{\text{prop. liveness info. (Law 20)}\} \\
\quad ((VL1i, coVL1i := V1, coV1 ; S')][\text{live VLf} ; (V := VLf)]][\text{live } V] \\
= \{\text{prop. liveness info. (Law 20)}\} \\
\quad ((VL1i, coVL1i := V1, coV1 ; S'][\text{live VLf}])[\text{live VLf}] ; (V := VLf)][\text{live } V] \\
\subseteq \{SV' := \text{fi-slice. } S'. VLf (Q1 \text{ of fi-slice})\} \\
\quad ((VL1i, coVL1i := V1, coV1 ; SV'][\text{live VLf}])[\text{live VLf}] ; (V := VLf)][\text{live } V] \\
= \{\text{prop. liveness info. (Law 20)}\} \\
\quad ((VL1i, coVL1i := V1, coV1 ; SV')[\text{live VLf} ; (V := VLf)][\text{live } V] \\
\quad \{Q1 \text{ of Transformation } D.6 \text{ Theorem 9.6 } DLs := \text{glob. } S' \setminus ND, \\
\quad SV := \text{fromSSA.} (SV', (V, coV), (VL1i, coVL1i), (VLf, coVLf), ND, DLs)\} \\
\quad ((SV ; VLf := V)[\text{live VLf}] ; (V := VLf)][\text{live } V] \\
= \{\text{prop. liveness info. (Law 20)}\} \\
\quad (SV ; (VLf := V) ; (V := VLf)][\text{live } V] \\
= \{\text{assignment-based sub. (Law 18): } VLf \circ V\} \\
\quad (SV ; (VLf := V) ; (V := V)][\text{live } V] \\
= \{\text{remove redundant self-assignment (Law 2)}\} \\
\quad (SV ; (VLf := V)][\text{live } V] \\
= \{\text{remove dead assignment (Law 24): } VLf \circ V\} \\
\quad SV[\text{live } V].\end{align*}\]

The success of \textit{fromSSA} (in the derivation above) depends on the validity of preconditions P1-P8 of Transformation \textbf{D.6} This is indeed guaranteed as is shown in the following.
9.2.2 An SSA-based slice is de-SSA-able

Let \( S' \) be the SSA version of a core statement \( S \). Then, the slicing algorithm, once operated on the set of slides \( \text{slides}\, S' \), is flow-sensitive. For such slices to be correct syntax-preserving slices, we need to show they are de-SSA-able.

**Theorem 9.6.** Any slide-independent statement from the SSA version of any core statement is de-SSA-able.

That is, let \( S \) be any core statement and let \( X, Y := (\text{def.}\, S), (\text{glob.}\, S \setminus \text{def.}\, S), X1 := (X \cap ((X \setminus \text{ddef.}\, S) \cup \text{input.}\, S)) \) and \( \text{def.}\, S' := \text{fresh.}\, ((X, Y)) \); let \( S' \) be the SSA version of \( S \), defined as \( S' := \text{toSSA.}(S, X, XLI, XLF, Y, (XLI, XLF)) \); let \( XLS := ((XLI, XLF) \cup (\text{def.}\, S' \setminus Y)) \) be the full set of instances (of \( X \), in \( S' \)) and let \( XLI \) be any (slide-independent) subset of those instances, with final instances \( XLI' := XLI \cap XLF \); finally let \( S' := \text{slides.}\, S'.XLI \) be the corresponding (slide-independent) statement; then \( S' \) is de-SSA-able. That is, all preconditions, P1-P8, of the \text{fromSSA} algorithm hold for \( SI := \text{fromSSA.}(S', X, XLI, XLF, XLS) \).

The proof can be found in Appendix D.

9.2.3 The refined algorithm

Following the derivation above, our SSA-based flow-sensitive slicing algorithm is given in Figure 9.2. A given program \( S \) is translated into its corresponding SSA version \( S' \); the flow-insensitive slice \( SV' \) of \( S' \) is taken with respect to final instances \( VLF \) of \( V \); finally, \( SV' \) is translated back from SSA by merging all instances.

The algorithm is correct in the sense that, for any core (and hence deterministic) statement \( S \) and set of variables \( V \), we get \( \text{slice.}\, S.V \) as a valid slice of \( S \) with respect to \( V \). That is, requirements Q1 and Q2 of slicing both hold.

The former \((S[\text{live}\, V] \subseteq (\text{slice.}\, S.V)[\text{live}\, V])\) follows from of the derivation above. Note that, indirectly, this property is a consequence of the corresponding Q1 of \( \text{fi-slice} \).

For the latter \((\text{def.}\,(\text{slice.}\, S.V) \subseteq \text{def.}\, S)\), we need to investigate the effects of \text{toSSA}, \( \text{fi-slice} \) and \text{fromSSA} on defined variables. Let \( SV := \text{slice.}\, S.V \) and let \( Vd := V \cap \text{def.}\, S \) such that \( \text{def.}\, S = (Vd, \text{coV}) \). Thus we need to show \( \text{def.}\, SV \subseteq (Vd, \text{coV}) \).

Firstly, since \( \text{def.}\, S = (Vd, \text{coV}) \), we observe \( \text{def.}\, S' \) involves instances of \( (Vd, \text{coV}) \) exclusively. Secondly, Q2 of \( \text{fi-slice} \) ensures \( \text{def.}\, SV' \subseteq \text{def.}\, S' \). Finally, since all instances of \( (Vd, \text{coV}) \) in \( SV' \) are successfully merged (see Q2 of \text{fromSSA} and Theorem 9.6), we end up with \( \text{def.}\, SV \subseteq (Vd, \text{coV}) \) as required.
Given a core statement \( S \) and variables of interest \( V \), compute the flow-sensitive slice, \( \text{slice}.S.V \), as follows:

\[
\text{slice}.S.V \triangleq \text{fromSSA}(SV', (V, coV), (VLi, coVLi), (VLf, coVLf), ND, DLs)
\]

where \( coV := \text{def}.S \setminus V \),

\[
SV' := \text{fi-slice}.S'.VLf,
S' := \text{toSSA}(S, (V, coV), (VLi, coVLi), (VLf, coVLf), ND, (VLi, coVLi, VLf, coVLf)),
V1, coV1 := (V \cap \text{input}.S), (coV \cap \text{input}.S),
DLs := \text{glob}.S' \setminus ND,
(VLi, coVLi, VLf, coVLf) := \text{fresh}((V1, coV1, V, coV), \text{glob}.S)
\]

and \( ND := \text{glob}.S \setminus (V, coV) \).

Figure 9.2: An SSA-based flow-sensitive slicing algorithm.

### 9.2.4 Example

In contrast to the flow-insensitive slicer (as demonstrated in Section 9.1.2), this refined algorithm would yield an accurate slice for \( sum \), as is shown here:

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

The first step of the algorithm is to turn the program into SSA form:
At this point, the flow-insensitive algorithm slices the middle part above for the final instance \( \text{sum}_2 \) of \( \text{sum} \). The slide of \( \text{sum}_2 \) depends on the slides of \( \text{sum}_1, i_2 \) and \( \text{sum}_3 \); \( i_2 \) in turn depends on \( \{i_1, i_2, i_3\} \) whereas \( \text{sum}_3 \) depends on \( \{i_2, \text{sum}_2\} \). We thus get \( \{i_1, i_2, i_3, \text{sum}_1, \text{sum}_2, \text{sum}_3\} \) as the reflexive transitive closure of slide dependence on \( \text{sum}_2 \), hence the requested slide-independent set \( \{\text{sum}_2\}^* \), yielding the following program:

```
| [var i_1, i_2, i_3, i_4, i_5, i_6, sum_1, sum_2, sum_3, prod_4, prod_5, prod_6
 ; i_1, sum_1 := 0, 0
 ; i_2, sum_2 := i_1, sum_1
 ; while i_2 < a.length do
   ; i_3, sum_3 := i_2 + 1, sum_2 + a[i_2]
   ; i_2, sum_2 := i_3, sum_3
 od
 ; i_4, prod_4 := 0, 1
 ; i_5, prod_5 := i_4, prod_4
 ; while i_5 < a.length do
   ; i_6, prod_6 := i_5 + 1, prod_5 * a[i_5]
   ; i_5, prod_5 := i_6, prod_6
 od
 ; i, sum, prod := i_5, sum_2, prod_5
]
```

Returning from SSA would then, as desired, yield
\[ i, \text{sum} := 0,0 \\
; \text{while} \ i < \text{a.length} \text{ do} \\
\quad i, \text{sum} := i+1, \text{sum+a[i]} \\
\text{od} \]

Similarly, requesting the slice of \textit{prod}, from this SSA-based slicing algorithm, would identify the second loop alone, whereas the earlier naive algorithm would unnecessarily add the first loop (for \textit{i}).

Indeed, the example above was specifically chosen to highlight the differences between our naive flow-insensitive slicer and the SSA-based one. Nevertheless, it should be noted that had the two loops been tangled as one, the slicer (in fact both slicers) would still identify the desired, accurate slice.

9.3 Slice extraction revisited

9.3.1 The transformation

With the SSA-based flow-sensitive slicing algorithm, we are for the first time in position to constructively express a sliding transformation. We base the transformation on Refinement 7.4 and produce the slice of the variables for extraction as the extracted code and the slice of the remaining variables as the complement.

Transformation 9.7. Let \( S \) be any core statement and let \( V \) be any (user selected) set of variables to be extracted; then

\[
S \sqsubseteq \Box \begin{array}{l}
\quad \text{\texttt{var } iV, icoV, fV; } iV, icoV := V', coV \\
\quad ; SV \\
\quad ; fV := V' \\
\end{array} \\
\mid \begin{array}{l}
\quad V', coV := iV, icoV \\
\quad ; ScoV \\
\quad ; V' := fV \end{array} \]

where \( V' := V \cap \text{def}.S, \)

\[
\quad coV := \text{def}.S \setminus V', \\
(iV, icoV, fV) := \text{fresh}.((V', coV, V'), (V \cup \text{glob}.S)), \\
SV := \text{slice}.S.V' \\
\quad \text{and } \quad ScoV := \text{slice}.S.coV 
\]

Proof.

\[
S
\]
9.3.2 Evaluation and discussion

With the above transformation, for example, the computation of `sum` in the scenario of Section 7.1 can be correctly untangled from that of `prod`, as desired.

Our current approach has been inspired by the tucking transformation [10]. In comparing Tuck to sliding, we first observe that global variables are inherently unsupported by Tuck, and whenever a live-on-exit variable is defined in both the extracted slice and its complement, the transformation has to be rejected. For example, when untangling `sum` and `prod` as was just recalled from Section 7.1, the loop variable `i` is defined in both loops of the resulting program; had the final value of `i` been used after the loop (e.g. in computing the average), Tuck would have been rejected.

Our semantic framework, in contrast, is expressive enough to avoid such limitations. The importance of this improvement over tucking is highlighted by the observation that in the presence of global variables, and in order to avoid the need for full program analysis (i.e. beyond the context of extraction), one has to assume all those variables are, indeed, live-on-exit.

Another notable difference between Tuck and sliding is in the construction of the complement. Their complement is the slice from all non-extracted statements, whereas we slice from the end of scope, on all non-extracted variables. This approach has been inspired by Gallagher's view of a program as a union of slices [22]. There, a program maintenance process, based on that view, is formalised along with the dependences between various slices. Consequently, conditions are derived for detecting non-interference of changes on a set of slices. Thus, some changes, e.g. when debugging, can be performed on a subprogram — leaving the merge of those changes in the
full program to an accompanying tool — with such confidence that eliminates the need for e.g. reduction testing.

In comparing Tuck and sliding’s approaches, we note that on the one hand, their complement would include slices from dead statements, if present. This in turn might lead to unnecessary duplication and possible rejection. Since the slice of all defined variables on a given statement, as in Gallagher’s view, will never include such dead code, its presence would have no affect on our approach.

On the other hand, Tuck’s complement has the potential of being more accurate than that of the current version of sliding. This might be the case whenever any possibly-final-definition of a non-extracted variable \( y \) (i.e. a definition that may reach the end of scope) is included in the extracted code. Tuck’s complement will include such a definition only if it is indirectly relevant for other non-extracted statements, whereas ours would definitely include it.

In order to understand the implications of this problem, we further investigate it, distinguishing two cases. Firstly, if all possibly-final-definitions of such a variable, \( y \), are extracted, the full slice on \( y \) is guaranteed to be included in the extracted code. In such cases, we solve the problem by including \( y \) in the set of extracted variables (see Chapter 12, where an optimal sliding transformation is sought).

Secondly, in cases where at least one such definition of \( y \) was extracted, and at least one other definition was not, we must distinguish two sub-cases. If \( y \) is live-on-exit, they will reject the transformation. Otherwise, their complement may indeed be smaller, since we assume all variables are live-on-exit. Accordingly, our sliding transformation may benefit from deriving a simple corollary for the case in which the live-on-exit variables are explicitly given, in which case, the complement should be composed of the slice of all non-extracted and live-on-exit variables.

We conclude that our results so far enjoy Tuck’s untangling ability with improved applicability and comparable levels of duplication. Further reduction in such duplication is still possible, as is explained next.

A valid criticism (of both Tuck and our current version of sliding) is that the levels of code duplication are potentially too high. This is highlighted by Komondoor, in his PhD thesis [39]. For example, if the extracted variable’s final value is used in the complement, the entire slice would be duplicated.

As was explained in Chapter 2, Komondoor’s alternative solutions (KH00 and KH03 with Horwitz [38] [39] as well as a variation on KH03 in his PhD thesis [37]), were not designed for untangling by slice extraction and are hence not applicable in our immediate context. Nevertheless, ideas from his approach have inspired and contributed to our decomposition of slides and the corresponding improvements to sliding, as will be explored shortly.
9.4 Summary

This chapter has developed a slicing algorithm, based on the observation that slide-independent sets of slides (from the previous chapter) yield correct slices. The SSA-based slicing algorithm has allowed a constructive formulation of a first sliding transformation, based on the refinement rule of semantic slice extraction (from two chapters back).

This version of sliding has been shown to enjoy Tuck’s untangling abilities, with improved applicability, while suffering, as Tuck, from potential over-duplication. Reductions in such duplication will be explored and formalised in the next chapter.
Chapter 10

Co-Slicing: Advanced Duplication Reduction in Sliding

In the preceding chapters, a basic slice-extraction refactoring has been introduced. Its automation through a sliding transformation has been discussed, along with correctness issues. The problematic duplication of the whole program in scope, as of our initial formulation, has been followed by slicing both the extracted code and its complement, in an attempt to reduce code duplication.

However, the levels of duplication introduced by sliding, so far, are still too high. This is so in cases where both the slice and its complement share some computation, but instead of reusing this computation’s extracted result in the complement, the computation’s code ends up being duplicated.

In this chapter, an advanced sliding strategy for reducing the levels of duplication is proposed, formalised and applied to sliding.

10.1 Over-duplication: an example

As an example of over-duplication, consider the following sliding of \textit{sum}. 

\[
i,\text{sum},\text{prod} := 0,0,1
; \text{while } i < \text{a.length} \text{ do}
\]
\[
i,\text{sum},\text{prod} :=
\]
\[
i+1,\text{sum}+\text{a}[i],\text{prod}+\text{a}[i]
\]
\[
od
; \text{out} \ll \text{sum}
; \text{out} \ll \text{prod}
\]

In applying Transformation 9.7 with \( V := \{\text{sum}\} \), we note that the whole extracted code ends up being duplicated in the complement, unnecessarily.

\[
|\text{var } ii,\text{isum},\text{iprod},\text{iout},\text{fsum}
; \text{ii, isum, iprod, iout := i, sum, prod, out}
; \text{i, sum} := 0,0
; \text{while } i < \text{a.length} \text{ do}
\]
\[
i, \text{sum} :=
\]
\[
i+1, \text{sum}+\text{a}[i]
\]
\[
od
; \text{fsum} := \text{sum}
\]
\[
|\text{i, sum, prod, out := ii, isum, iprod, iout}
; \text{i, sum, prod} := 0,0,1
; \text{while } i < \text{a.length} \text{ do}
\]
\[
i, \text{sum, prod} :=
\]
\[
i+1, \text{sum}+\text{a}[i], \text{prod}+\text{a}[i]
\]
\[
od
; \text{out} \ll \text{sum}
; \text{out} \ll \text{prod}
; \text{sum} := \text{fsum}
|
\]

10.2 Final-use substitution

We propose to further reduce duplication in sliding through what we call final-use substitution. A final use is a reference to a variable’s value (e.g. \text{sum} in the example above), in a program point where it is guaranteed to hold its final value (e.g. where \text{sum} is appended to \text{out}).

If the slice of the variable under discussion has been extracted through sliding, that final value might be available in the complement (e.g. in backup variable \text{fsum}), saving us the need to recompute it.
10.2.1 Example

To demonstrate the workings of final-use substitutions, we return to the example (from above) of computing and printing the sum and product of an array of integers. As was already mentioned, trying to extract \textit{sum} using our current proposed transformation would lead to duplication of the whole extracted slice, unnecessarily.

One way of avoiding that duplication is to use \textit{fsum} in the complement.

```
| [var ii,isum,iprod,iout,fsum
; ii,isum,iprod,iout:=i,sum,prod,out
; i,sum,prod := 0,0,1
; while i<a.length do
  i,sum,prod :=
  i+1,sum+a[i],prod*a[i]
  od
; out << sum
; out << prod
| ; fsum:=sum
}
```

```
| i,sum,prod,out:=ii,isum,iprod,iout
; i,sum,prod := 0,0,1
; while i<a.length do
  i,sum,prod :=
  i+1,sum+a[i],prod*a[i]
  od
; out << fsum
; out << prod
| ; sum:=fsum
```

Note that not all uses of \textit{sum} were replaced with \textit{fsum}; the use inside the loop is not of a final value, and must not be replaced.

As before (in Transformation 9.7), the above version of statement duplication, with final-use substitution, has the potential of introducing dead code, which can subsequently be removed. At this point, slicing (for \{\textit{sum}\} in the extracted code and \{\textit{i, prod, out}\} in the complement), would successfully remove the repeated computation of \textit{sum}; leading to:
10.2.2 Deriving the transformation

Final-use substitution can be formalised in the following way. Starting with “$S; \{V = fV\}$” where $fV \circ \text{glob}.S$ we transform $S$ into $S' := S[\text{final-use } V \setminus fV]$ demanding “$S; \{V = fV\}$” = “$S'; \{V = fV\}$”. Statement $S'$ will be using variables in $fV$ instead of $V$ in points to which the corresponding assertion can be propagated.

The full derivation of $S[\text{final-use } V \setminus fV]$ is given in Appendix E; the resulting transformation is given in Figure 10.1.

With final-use substitution constructively defined, we now turn to derive an advanced solution to slice extraction via sliding.

10.3 Advanced sliding

10.3.1 Statement duplication with final-use substitution

In the following, we show that any core statement $S$ is equivalent to its duplicated version, in which the computation of variables ($V_r, V_{nr}$) is separated from that of the complementary set $coV$ (such that $\text{def}.S = (V_r, V_{nr}, coV)$), and variables $V_r$ are offered for reuse in the complement, through the backup variables of their final values, $fV_r$.

**Program equivalence 10.1.** Let $S, V_r, V_{nr}, coV, iV_r, iV_{nr}, icoV, fV_r, fV_{nr}$ be a core statement and eight sets of variables, respectively; then

```plaintext
; fsum := sum
```

```plaintext
; i, prod := 0, 1
; while i < a.length do
  i, prod :=
  i + 1, prod * a[i]
  od
; out << fsum
; out << prod
; sum := fsum
```

Let $S$ be a core statement and $X$ a set of variables, to be substituted by a corresponding set $X'$ of fresh variables; the final-use substitution of $S$ on $X$ with $X'$ is defined, by cases of $S$, as follows:

- \[(X_1, Y := E_1, E_2)[\text{final-use } X_1, X_2 \setminus X'_1, X'_2] \triangleq \]
  - “ $X_1, Y := E_1[X_2 \setminus X'_2], E_2[X_2 \setminus X'_2]$ ” where $X = (X_1, X_2)$, $X \circ Y$ and $X' = (X'_1, X'_2)$ ;

- \[(S_1 ; S_2)[\text{final-use } X_1, X_2 \setminus X'_1, X'_2] \triangleq \]
  - “ $S_1[\text{final-use } X_2 \setminus X'_2] ; S_2[\text{final-use } X_1, X_2 \setminus X'_1, X'_2]$ ” where $X_1 := X \cap \text{def}.S_2$, $X_2 := X \setminus X_1$ and $X'_1, X'_2$ are the corresponding subsets of $X'$ ;

- \[(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi})[\text{final-use } X_1, X_2 \setminus X'_1, X'_2] \triangleq \]
  - “ if $B[X_2 \setminus X'_2]$ then $S_1[\text{final-use } X_1, X_2 \setminus X'_1, X'_2]$ 
    else $S_2[\text{final-use } X_1, X_2 \setminus X'_1, X'_2]$ fi ” where $X_1 := X \cap \text{def}.(S_1, S_2)$, $X_2 := X \setminus X_1$ and $X'_1, X'_2$ are the corresponding subsets of $X'$ ; and

- \[(\text{while } B \text{ do } S_1 \text{ od})[\text{final-use } X_1, X_2 \setminus X'_1, X'_2] \triangleq \]
  - “(while $B[X_2 \setminus X'_2]$ do $S_1[\text{final-use } X_2 \setminus X'_2]$ od) where $X_1 := X \cap \text{def}.S_1$, $X_2 := X \setminus X_1$ and $X'_2$ is the subset of $X'$ corresponding to $X_2$ .

Figure 10.1: An algorithm of final-use substitution.
Proof.

\[
S = \begin{array}{c}
(i_{Vr}, i_{Vnr}, i_{coV} := Vr, Vnr, coV) \\
; S \\
; f_{Vr}, f_{Vnr} := Vr, Vnr
\end{array}
\begin{array}{c}
Vr, Vnr, coV := i_{Vr}, i_{Vnr}, i_{coV} \\
; S[\text{final-use } Vr \setminus f_{Vr}] \\
; Vr, Vnr := f_{Vr}, f_{Vnr}[\text{live } Vr, Vnr, coV]
\end{array}
\]

provided \( def.S = (Vr, Vnr, coV) \)

and \((i_{Vr}, i_{Vnr}, i_{coV}, f_{Vr}, f_{Vnr}) \circ \text{glob}.S \).

\[
S = \{
\begin{array}{l}
\text{prepare for statement duplication (Lemma 6.2) with } \\
V, i_{V}, f_{V} := (Vr, Vnr), (i_{Vr}, i_{Vnr}), (f_{Vr}, f_{Vnr}); \\
\text{provisos } def.S = (Vr, Vnr, coV) \text{ and } (i_{Vr}, i_{Vnr}, i_{coV}, f_{Vr}, f_{Vnr}) \circ \text{glob}.S
\end{array}
\]

\[
\begin{array}{l}
(i_{Vr}, i_{Vnr}, i_{coV} := Vr, Vnr, coV) ; S ; f_{Vr}, f_{Vnr} := Vr, Vnr \\
\{Vr = f_{Vr}\} ; Vr, Vnr := f_{Vr}, f_{Vnr}[\text{live } Vr, Vnr, coV]
\end{array}
\]

\[
= \{
\begin{array}{l}
\text{intro. following assertion (Law 7) with } X, Y, E1, E2 := f_{Vnr}, f_{Vr}, Vnr, Vr; \\
(f_{Vr}, f_{Vnr}) \circ Vr \text{ due to both provisos and RE5}
\end{array}
\]

\[
\begin{array}{l}
(i_{Vr}, i_{Vnr}, i_{coV} := Vr, Vnr, coV) ; S ; f_{Vr}, f_{Vnr} := Vr, Vnr \\
\{Vr = f_{Vr}\} ; Vr, Vnr := f_{Vr}, f_{Vnr}[\text{live } Vr, Vnr, coV]
\end{array}
\]

\[
= \{
\begin{array}{l}
\text{statement duplication (Lemma 6.3) with } \\
V, i_{V}, f_{V} := (Vr, Vnr), (i_{Vr}, i_{Vnr}), (f_{Vr}, f_{Vnr}); \text{ provisos}
\end{array}
\]

\[
\begin{array}{l}
(i_{Vr}, i_{Vnr}, i_{coV} := Vr, Vnr, coV) ; S ; f_{Vr}, f_{Vnr} := Vr, Vnr \\
\{Vr = f_{Vr}\} ; Vr, Vnr := f_{Vr}, f_{Vnr}[\text{live } Vr, Vnr, coV]
\end{array}
\]

\[
= \{
\begin{array}{l}
\text{final-use sub.: correct by construction (see Figure 10.1 and Appendix E) and } \\
\text{variables } f_{Vr} \text{ are indeed fresh (proviso } f_{Vr} \circ \text{glob}.S)\}
\end{array}
\]

\[
\begin{array}{l}
(i_{Vr}, i_{Vnr}, i_{coV} := Vr, Vnr, coV) ; S ; f_{Vr}, f_{Vnr} := Vr, Vnr \\
\{Vr = f_{Vr}\} ; Vr, Vnr := f_{Vr}, f_{Vnr}[\text{live } Vr, Vnr, coV]
\end{array}
\]

\[
= \{
\begin{array}{l}
\text{remove aux. assertion (Lemma 10.2 see below) with } \\
V, i_{V}, f_{V} := (Vr, Vnr), (i_{Vr}, i_{Vnr}), (f_{Vr}, f_{Vnr})
\end{array}
\]

\[
\begin{array}{l}
(i_{Vr}, i_{Vnr}, i_{coV} := Vr, Vnr, coV) ; S ; f_{Vr}, f_{Vnr} := Vr, Vnr \\
\{Vr = f_{Vr}\} ; Vr, Vnr := f_{Vr}, f_{Vnr}[\text{live } Vr, Vnr, coV]
\end{array}
\]
**Lemma 10.2.** Let $S$ be any core statement with $\text{def}.S = (V, coV)$, $V_r \subseteq V$ (and $f_{V_r}$ the corresponding subset of $fV$) and $(iV, icoV, fV) \odot \text{glob}.S$; we then have

| “$iV, icoV := V, coV$” | $V, coV := iV, icoV$ |
| $; S$ | $; S[\text{final-use } V_r \setminus f_{V_r}]$ |
| $; fV := V$ | $; \{ V_r = f_{V_r} \}$ |

$=$

| “$iV, icoV := V, coV$” | $V, coV := iV, icoV$ |
| $; S$ | $; S[\text{final-use } V_r \setminus f_{V_r}]$ |
| $; fV := V$ | “$V_r, Vnr, coV : iV, iVnr, icoV$ $;$ $V_r, Vnr : fV_r, fV_{nr}[\text{live } V_r, Vnr, coV]$” |

The proof of that lemma is given in Appendix E.

### 10.3.2 Slicing after final-use substitution

The statement duplication with final-use substitution, as in the program equivalence above, can now be followed up by slicing, as was done earlier in Chapter 7.

**Refinement 10.3.** Let $S, SV, ScoV, V_r, V_{nr}, coV, iV_r, iV_{nr}, icoV, fV_r, fV_{nr}$ be three core statements and eight sets of variables, respectively; then

| “$(iV_r, iV_{nr}, icoV := V_r, V_{nr}, coV)$” | $V_r, Vnr, coV := iV_r, iV_{nr}, icoV$ |
| $; SV$ | $; ScoV$ |
| $; fV_r, fV_{nr} := V_r, Vnr$ | $; V_r, Vnr := fV_r, fV_{nr}[\text{live } V_r, Vnr, coV]$ ” |

provided $\text{def}.S = (V_r, Vnr, coV)$,

- $S[\text{live } V_r, Vnr] \subseteq SV[\text{live } V_r, Vnr]$,
- $S[\text{final-use } V_r \setminus f_{V_r}[\text{live } coV] \subseteq ScoV[\text{live } coV]$,
- $\text{def}.SV \subseteq \text{def}.S$,
- $\text{def}.ScoV \subseteq \text{def}.S$ and
- $(iV, icoV, fV) \odot \text{glob}.S$.

**Proof.**

$S$
CHAPTER 10. CO-SLICING

= \{\text{duplicate statement (Program equivalence 10.1)}:
    \text{being a core statement, } S \text{ is indeed deterministic,}
    \text{def}. S = (V_r, V_{nr}, coV) \text{ and } (iV_r, iV_{nr}, icV, fV_r, fV_{nr}) \circ \text{glob}. S \text{ (proviso)}\}

\{\text{Refinement 7.5 with}
    \text{def}. S[\text{final-use } V_r \setminus fV_r] = \text{def}. S \text{ since only uses are replaced by final-use sub.}\}

\{\text{Refinement 7.5 with}
    \text{def}. S[\text{final-use } V_r \setminus fV_r] = \text{def}. S \text{ since only uses are replaced by final-use sub.}\}

10.3.3 Definition of co-slicing

Observing the requirements $S[\text{final-use } V_r \setminus fV_r][\text{live } coV] \subseteq ScoV[\text{live } coV]$ and $\text{def}. ScoV \subseteq \text{def}. S$ above, we formalise co-slices as follows:

\textbf{Definition 10.4} (Complement-Slice (or Co-Slice)). Let $S$ be a core (and hence deterministic) statement and $V$ be a set of variables for extraction; let $V_r$ be a subset of $V$ to be made reusable through fresh variables $fV_r$. Any statement $ScoV$ for which the two requirements

(\text{Q1:}) $S[\text{final-use } V_r \setminus fV_r][\text{live } coV] \subseteq ScoV[\text{live } coV]$ and $\text{def}. ScoV \subseteq \text{def}. S$

both hold, is a correct co-slice of $S$ with respect to $V, V_r$ and $fV_r$.

A co-slicing algorithm (see Figure 10.2) is consequently derived from the above definition and the corresponding properties Q1 and Q2 of slicing. From those properties, the algorithm’s correctness follows.

10.3.4 The sliding transformation

The refinement rule from above, along with the formal definition of co-slices and the corresponding constructive co-slicing algorithm, are now combined in yielding an advanced sliding transformation.

\textbf{Transformation 10.5.} Let $S$ be any core statement and let $V_r, V_{nr}$ be any two disjoint (user selected) sets of variables to be extracted, with $V_r$ to be made available for reuse in the complement; then
Let $S, V, V_r, fV_r$ be a core statement and three sets of variables, respectively. The function co-slice for statement $S$ with respect to $V, V_r, fV_r$, is defined as follows:

\[
\text{co-slice}.S, V, V_r, fV_r \triangleq \text{slice}.S[\text{final-use } V_r \setminus fV_r].coV
\]

where $coV := \text{def}.S \setminus V$

provided $V_r \subseteq V$

and $fV_r \circ (V \cup \text{glob}.S)$.

Proof.

Figure 10.2: A co-slicing algorithm, based on slicing and final-use substitution.

\[
\begin{align*}
S \subseteq & \quad \{ \text{Refinement } 10.3 \} : \quad (V_r', V_nr', coV) = \text{def}.S \text{ by def. of } V_r', V_nr', coV; \\
& \quad S[\text{live } V_r', V_nr'] \subseteq SV[\text{live } V_r', V_nr'] \text{ by Q1 of slice; } \\
& \quad \text{def}SV \subseteq \text{def}S \text{ by Q2 of slice; similarly} \\
& \quad S[\text{live } coV] \subseteq ScoV[\text{live } coV] \text{ by Q1 of co-slice; } \\
& \quad \text{def}ScoV \subseteq \text{def}S \text{ by Q2 of co-slice} \\
& \quad " (iV_r, iV_nr, icoV := V_r', V_nr', coV ; SV ; fV_r, fV_nr := V_r', V_nr') \\
& \quad V_r', V_nr', coV := iV_r, iV_nr, icoV ; ScoV ; V_r', V_nr' := fV_r, fV_nr) \\
& \quad [\text{live } V_r', V_nr', coV] "
\end{align*}
\]
\[ \{ \text{def. of live: } (\text{def.SV} \cup \text{def.ScoV}) \subseteq (V', coV) \text{ (again, Q2 of slice and co-slice)} \]
and \((iVr, iVnr, icoV, fVr, fVnr) \odot (Vr', Vnr', coV)\} \]

“ \[\{\text{var } iVr, iVnr, icoV, fVr, fVnr; \]
iVr, iVnr, icoV := Vr', Vnr', coV ; SV ; fVr, fVnr := Vr', Vnr' ;
Vr', Vnr', coV := iVr, iVnr, icoV ; ScoV ; Vr', Vnr' := fVr, fVnr\} \]

10.4 Summary

This chapter has introduced an advanced sliding transformation in which the complement reuses a selection of extracted results, thus yielding a potentially smaller complement, or as we call it co-slice. Co-slicing has been formalised through a so-called final-use substitution. Constructive definitions of that substitution, and hence of a co-slicing algorithm, have been developed.

In comparison to our earlier sliding transformation, the advanced version potentially duplicates less code. However, the price takes the form of extra compensatory code. This is due to final-use substitution, renaming some used variables in the complement. This renaming, to which we are opposed, in general, in an attempt to keep the resulting program as close to the original as possible, has been introduced in order to avoid name clashing. However, since our co-slicing algorithm involves the removal of dead code by slicing, after final-use substitution, some renaming can potentially be undone.

The elimination of redundant compensatory code, after sliding, will be pursued in the next chapter. In particular, undoing the renaming of final-use substitution will be formalised, thus yielding the concepts of compensation-free co-slices and compensation-free sliding.
Chapter 11

Penless Sliding

When sliding is expected to maintain all variable names (i.e. no renaming), it is not the case that any final-use substitution yields a valid co-slice. The notion of compensation-free (or penless) co-slice is introduced in this chapter. Moreover, a general improvement of sliding by eliminating redundant backup variables is explored, ultimately leading to the formulation of (the conditions for) a completely penless sliding transformation. The elimination of backup variables is based on a liveness-analysis related approach to variable merging, on the lines of our merge-vars algorithm (Appendix D), which has been applied for the return from SSA (in Section 8.6.2).

11.1 Eliminating redundant backup variables

We begin by detecting and eliminating redundant backup variables. When sliding variables $(V_r, V_{nr})$ away from $coV$ on statement $S$ (with $def.S \subseteq (V_r, V_{nr}, coV)$ and with $V_r$ available for reuse in the complement), we naively introduce backup variables $(iV_r, iV_{nr}, icoV)$ for initial values and $fV_r, fV_{nr}$ for final value of extracted variables.

However, some of those backup variables might in fact be redundant and should hence be removed.

11.1.1 Motivation

Why should those be removed? Following practical considerations, we note that such backup would require an unnecessarily large storage space, and the two operations of making the backup and retrieving it would have an unwanted impact on execution time.

Furthermore, suppose we waive our language assumption that any variable is cloneable, and in turn strengthen sliding’s preconditions to ensure no uncloneable variable is actually cloned. In that context, the removal of redundant backup variables will be crucial for the applicability of
sliding.

Finally, as was hinted above, when renaming of variables in the complement must be avoided, the removal of redundant backup variables will allow such renaming to be undone.

### 11.1.2 Example

Recall the co-slicing example from the preceding chapter (Section 10.2.1). There, asking to extract and reuse the variable *sum* (i.e. when $V_r, V_{nr}, coV := \{sum\}, \emptyset, \{i, prod, out\}$ in applying Transformation 10.5) from the program of Section 10.1 gave us the following result:

$$
\begin{align*}
&| [\text{var } ii, isum, iprod, iout, fsum] \\
&; ii, isum, iprod, iout := i, sum, prod, out \\
&; i, sum := 0, 0 \\
&; \text{while } i < a.\text{length} \text{ do} \\
&\quad i, sum := \\
&\quad i + 1, sum + a[i] \\
&\quad od \\
&; fsum := sum \\
&| & i, sum, prod, out := ii, isum, iprod, iout \\
&; i, prod := 0, 1 \\
&; \text{while } i < a.\text{length} \text{ do} \\
&\quad i, prod := \\
&\quad i + 1, prod \times a[i] \\
&\quad od \\
&; \text{out } \ll fsum \\
&; \text{out } \ll prod \\
&; sum := fsum \\
&| \n\end{align*}
$$

Here, the sets of backup variables of initial values $iV_r, iV_{nr}, icoV$ are $\{isum\}, \emptyset, \{ii, iprod, iout\}$, respectively, and the backup of final values $fV_r, fV_{nr}$ is $\{fsum\}, \emptyset$, respectively. Which of the backup variables $\{ii, isum, iprod, iout, fsum\}$ is redundant?

### 11.1.3 Dead-assignments-elimination and variable-merging

We remove redundant backup variables by combining dead-assignments-elimination and the merging of such backup variables with their corresponding original variables. Recall our *merge-vars* algorithm (as mentioned for returning from SSA, see Section 8.6.2 and Appendix D). According to that approach, merging members of $iV_r, iV_{nr}, icoV$ with corresponding members of $V_r, V_{nr}, coV$ is possible if they are never simultaneously-live, never defined in the same assignment, and one is never defined in an assignment where the other is live-on-exit.

Definition on the same assignment (e.g. of *sum* and its backup *isum* or *fsum*) is not possible after sliding, since the backup variables are defined only in designated statements. Furthermore, since we precede this step by dead-assignments-elimination, cases of def-on-live may only occur
in conjunction with simultaneous liveness; the defined variable must be live too, or its definition would have been removed. (Note that at this stage, the dead-assignments-elimination removes unused backup variables of initial values only, e.g. \{ii, isum, iprod\} above, but not iout, since the retrieval from backup of all final values, e.g. the sum := fsum above, renders such backup, e.g. fsum, live at its point of initialisation, e.g. the fsum := sum above.)

So it is simultaneous liveness that we should worry about. Backup variables for initial values (e.g. iout, all the others have already been removed) are alive from entry to the extracted slice all the way to the exit from the initialisation of backup of final values. There, the defined extracted variables \(V'\) (e.g. \{sum\}) are used and their corresponding live initial backup variables (none of those in our example, since isum is gone) must remain, as they are simultaneously-live on exit from the extracted slice. On the other hand, members of coV (e.g. \{i, prod, out\}) are alive in the extracted slice SV only if in glob.SV. Thus, the backup variables for non-extracted initial variables that do not occur free in the extracted slice can be merged with the corresponding original variables. Hence, in our example, out and iout can be merged.

Initially, after sliding, the backup variables of all final values (e.g. fsum) are live-on-exit from the complement, and hence also when initialised, but the corresponding program variables (i.e. members of \(V'\), e.g. sum) are not, since they are defined there). If those are neither used nor defined in the complement, the need for backup disappears. That is, backup variables of final value of \(V' \setminus glob.ScoV\) should be merged with the corresponding original variables. In our example, hence, sum can be merged with fsum.

The following is the resulting sliding refinement rule, after eliminating redundant backup variables.

**Refinement 11.1.** Let \(S, SV, ScoV, Vr, Vnr, coV, iVr, iVnr, icoV, fVr, fVnr\) be three core statements and eight sets of variables, respectively; then

\[
\begin{align*}
S \sqsubseteq & \quad \begin{aligned}
& (iVr1, iVnr1, icoV11 := Vr1, Vnr1, coV11)
\end{aligned} \\
& \quad ; SV \\
& \quad ; fVr1, fVr2, fVnr1, fVnr2 := Vr1, Vr2, Vnr1, Vnr2 \\
\end{align*}
\]

\[
\begin{aligned}
& Vr1, Vnr1, coV11 := iVr1, iVnr1, icoV11 \\
& \quad ; ScoV[fVr3 \setminus Vr3] \\
& \quad ; Vr1, Vr2, Vnr1, Vnr2 := fVr1, fVr2, fVnr1, fVnr2)[live Vr, Vnr, coV]
\end{aligned}
\]
provided \( \text{def}.S = (V_r, V_{nr}, coV) \),
\[
S | \text{live } V_r, V_{nr} \subseteq SV | \text{live } V_r, V_{nr},
\]
\[
S | \text{final-use } V_r \setminus fV_r | [\text{live } coV] \subseteq ScoV | \text{live } coV,
\]
\[
\text{def}.SV \subseteq \text{def}.S,
\]
\[
\text{def}.ScoV \subseteq \text{def}.S,
\]
\[
(iV_r, iV_{nr}, icoV, fV_r, fV_{nr}) \circ \text{glob}.S,
\]
\[
(V_r1, V_r2, V_r3) =
\]
\[
(V_r \cap \text{input}.ScoV), (V_r \cap (\text{def}.ScoV \setminus \text{input}.ScoV)), (V_r \setminus \text{glob}.ScoV),
\]
with \((iV_r1, iV_r2, iV_r3)\) the corresponding subsets of \(iV_r\)
and with \((fV_r1, fV_r2, fV_r3)\) the corresponding subsets of \(fV_r\),
\[
(V_{nr1}, V_{nr2}, V_{nr3}) =
\]
\[
((V_{nr} \cap \text{input}.ScoV), (V_{nr} \cap (\text{def}.ScoV \setminus \text{input}.ScoV)), (V_{nr} \setminus \text{glob}.ScoV))
\]
with \((iV_{nr1}, iV_{nr2}, iV_{nr3})\) the corresponding subsets of \(iV_{nr}\)
and with \((fV_{nr1}, fV_{nr2}, fV_{nr3})\) the corresponding subsets of \(fV_{nr}\)
and
\[
(coV1, coV12, coV2) =
\]
\[
(coV \cap \text{def}.SV \cap \text{input}.ScoV), (coV \cap (\text{input}.ScoV \setminus \text{def}.SV)), (coV \setminus \text{input}.ScoV)
\]
with \((icoV11, icoV12, icoV2)\) the corresponding subsets of \(icoV\).

**Proof.**

\[
S \subseteq \{ \text{Refinement } [10.3] \}
\]

\[
\text{“ } (iV_r, iV_{nr}, icoV := V_r, V_{nr}, coV ; SV ; fV_r, fV_{nr} := V_r, V_{nr} ; V_r, V_{nr}, coV := iV_r, iV_{nr}, icoV ; ScoV ; V_r, V_{nr} := fV_r, fV_{nr}) [\text{live } V_r, V_{nr}, coV] \text{ “}
\]
\[
= \{ \text{liveness analysis: } V_r1, V_{nr1}, coV1 =
\]
\[
(V_r \cap \text{input}.ScoV), (V_{nr} \cap \text{input}.ScoV), (coV \cap \text{input}.ScoV) \}
\]

\[
\text{“ } (((iV_r, iV_{nr}, icoV := V_r, V_{nr}, coV ; SV ; fV_r, fV_{nr} := V_r, V_{nr} ; V_r, V_{nr}, coV := iV_r, iV_{nr}, icoV) [\text{live } fV_r, fV_{nr}, V_r1, V_{nr1}, coV1] ]
\]
\[
; ScoV) [\text{live } fV_r, fV_{nr}, coV] ; V_r, V_{nr} := fV_r, fV_{nr}) [\text{live } V_r, V_{nr}, coV] \text{ “}
\]
\[
= \{ \text{remove dead assignments, see below (big step 1))} \}
\]

\[
\text{“ } (((iV_r1, iV_{nr1}, icoV1 := V_r1, V_{nr1}, coV1 ; SV ; fV_r, fV_{nr} := V_r, V_{nr} ; V_r1, V_{nr1}, coV1 := iV_r1, iV_{nr1}, icoV1) [\text{live } fV_r, fV_{nr}, V_r1, V_{nr1}, coV1] ]
\]
\[
; ScoV ; V_r, V_{nr} := fV_r, fV_{nr}) [\text{live } V_r, V_{nr}, coV] \text{ “}
\]
\[
= \{(coV1, icoV1) = ((coV11, coV12), (icoV11, icoV12)) \}
\]
\[ (((iVr_1, iVnr_1, iCoV_{11}, iCoV_{12}) := Vr_1, Vnr_1, CoV_{11}, CoV_{12})
; SV ; fVr, fVnr := Vr, Vnr
; Vr_1, Vnr_1, CoV_{11}, CoV_{12} := iVr_1, iVnr_1, iCoV_{11}, iCoV_{12})
[ live fVr, fVnr, Vr_1, Vnr_1, CoV_{11} ]
; ScoV ; Vr, Vnr := fVr, fVnr) [ live Vr, Vnr, CoV ] \]

= \{ eliminate redundant backup of initial values, see below (big step 2) \}

\[ (((iVr_1, iVnr_1, iCoV_{11}) := Vr_1, Vnr_1, CoV_{11} ; SV ; fVr, fVnr := Vr, Vnr
; Vr_1, Vnr_1, CoV_{11} := iVr_1, iVnr_1, iCoV_{11}) [ live fVr, fVnr, Vr_1, Vnr_1, CoV_{11} ]
; ScoV ; Vr, Vnr := fVr, fVnr) [ live Vr, Vnr, CoV ] \]

= \{ remove liveness info.; ( Vr, Vnr, fVr, fVnr ) =

( ( Vr_1, Vr_2, Vr_3), ( Vnr_1, Vnr_2, Vnr_3), ( fVr_1, fVr_2, fVr_3), ( fVnr_1, fVnr_2, fVnr_3) ) \}

\[ (((iVr_1, iVnr_1, iCoV_{11}) := Vr_1, Vnr_1, CoV_{11} ; SV
; fVr_1, fVr_2, fVnr_1, fVnr_2 := Vr_1, Vr_2, Vnr_1, Vnr_2
; Vr_1, Vnr_1, CoV_{11} := iVr_1, iVnr_1, iCoV_{11} ; ScoV
; Vr_1, Vr_2, Vr_3, Vnr_1, Vnr_2, Vnr_3 := fVr_1, fVr_2, fVnr_1, fVnr_2, fVnr_3) [ live Vr, Vnr, CoV ] \]

= \{ eliminate redundant backup of final values, see below (big step 3) \}

\[ (((iVr_1, iVnr_1, iCoV_{11}) := Vr_1, Vnr_1, CoV_{11} ; SV
; fVr_1, fVr_2, fVnr_1, fVnr_2 := Vr_1, Vr_2, Vnr_1, Vnr_2
; Vr_1, Vnr_1, CoV_{11} := iVr_1, iVnr_1, iCoV_{11} ; ScoV[fVr_1 \backslash Vr_3]
; Vr_1, Vr_2, Vnr_1, Vnr_2 := fVr_1, fVr_2, fVnr_1, fVnr_2) [ live Vr, Vnr, CoV ] \]

\]

For big step 1 above, in which dead assignments are removed, we observe

\[ ( iVr, iVnr, iCoV := Vr, Vnr, CoV ; SV ; fVr, fVnr := Vr, Vnr
; Vr, Vnr, CoV := iVr, iVnr, iCoV) [ live fVr, fVnr, Vr_1, Vnr_1, CoV_{11} ] \]

= \{ remove dead assignments \}

\[ ( iVr, iVnr, iCoV := Vr, Vnr, CoV ; SV ; fVr, fVnr := Vr, Vnr
; Vr_1, Vnr_1, CoV_{11} := iVr_1, iVnr_1, iCoV_{11}) [ live fVr, fVnr, Vr_1, Vnr_1, CoV_{11} ] \]

= \{ liveness analysis \}

\[ ( ( iVr, iVnr, iCoV := Vr, Vnr, CoV) [ live iVr_1, iVnr_1, iCoV_{11} ]
; SV ; fVr, fVnr := Vr, Vnr
; Vr_1, Vnr_1, CoV_{11} := iVr_1, iVnr_1, iCoV_{11}) [ live fVr, fVnr, Vr_1, Vnr_1, CoV_{11} ] \]

= \{ remove dead assignments \}
“\((ivr1, i\text{Vnr1}, i\text{coV1} := \text{Vr1}, \text{Vnr1}, \text{coV1})[\text{live} \ i\text{Vr1}, i\text{Vnr1}, i\text{coV1}]\)
\(; SV ; f\text{Vr}, f\text{Vnr} := \text{Vr}, \text{Vnr}\)
\(; \text{Vr1}, \text{Vnr1}, \text{coV1} := ivr1, i\text{Vnr1}, i\text{coV1})[\text{live} f\text{Vr}, f\text{Vnr}, \text{Vr1}, \text{Vnr1}, \text{coV1}] \)
\(=\) \{remove liveness info.\}
“\((ivr1, i\text{Vnr1}, i\text{coV1} := \text{Vr1}, \text{Vnr1}, \text{coV1} \); \(SV ; f\text{Vr}, f\text{Vnr} := \text{Vr}, \text{Vnr}\)
\(; \text{Vr1}, \text{Vnr1}, \text{coV1} := ivr1, i\text{Vnr1}, i\text{coV1})[\text{live} f\text{Vr}, f\text{Vnr}, \text{Vr1}, \text{Vnr1}, \text{coV1}] \)

For big step 2 above, in which we eliminate redundant backup of initial values, we observe
“\((ivr1, i\text{Vnr1}, i\text{coV1} := \text{Vr1}, \text{Vnr1}, \text{coV1})[\text{live} f\text{Vr}, f\text{Vnr}, \text{Vr1}, \text{Vnr1}, \text{coV1}] \)
\(=\) \{split assignment: \((ivr1, i\text{Vnr1}, i\text{coV1}) \circ \text{coV12}\)\}
“\((ivr1, i\text{Vnr1}, i\text{coV11} := \text{Vr1}, \text{Vnr1}, \text{coV11} \); \(SV ; f\text{Vr}, f\text{Vnr} := \text{Vr}, \text{Vnr}\)
\(; \text{Vr1}, \text{Vnr1}, \text{coV11}, \text{coV12} := ivr1, i\text{Vnr1}, i\text{coV11}, i\text{coV12})[\text{live} f\text{Vr}, f\text{Vnr}, \text{Vr1}, \text{Vnr1}, \text{coV11}, \text{coV12}] \)
\(=\) \{swap statements: \(i\text{coV12} \circ (\text{glob.SV} \cup (f\text{Vr}, f\text{Vnr}, \text{Vr}, \text{Vnr}))\) and \(\text{(def.SV, fVr, fVnr)} \circ (i\text{coV12}, \text{coV12})\)\}
“\((ivr1, i\text{Vnr1}, i\text{coV11} := \text{Vr1}, \text{Vnr1}, \text{coV11}) \); \(SV ; f\text{Vr}, f\text{Vnr} := \text{Vr}, \text{Vnr}\)
\(; i\text{coV12} := \text{coV12} ; \; \text{Vr1}, \text{Vnr1}, \text{coV11}, \text{coV12} := ivr1, i\text{Vnr1}, i\text{coV11}, i\text{coV12})[\text{live} f\text{Vr}, f\text{Vnr}, \text{Vr1}, \text{Vnr1}, \text{coV11}, \text{coV12}] \)
\(=\) \{assignment-based sub.: \(\text{coV12} \circ i\text{coV12}\)\}
“\((ivr1, i\text{Vnr1}, i\text{coV11} := \text{Vr1}, \text{Vnr1}, \text{coV11}) \); \(SV ; f\text{Vr}, f\text{Vnr} := \text{Vr}, \text{Vnr}\)
\(; i\text{coV12} := \text{coV12} ; \; \text{Vr1}, \text{Vnr1}, \text{coV11}, \text{coV12} := ivr1, i\text{Vnr1}, i\text{coV11}, i\text{coV12})[\text{live} f\text{Vr}, f\text{Vnr}, \text{Vr1}, \text{Vnr1}, \text{coV11}, \text{coV12}] \)
\(=\) \{remove redundant self-assignment;
remove dead assignment: \(i\text{coV12} \circ (f\text{Vr}, f\text{Vnr}, i\text{Vr1}, i\text{Vnr1}, i\text{coV11})\)\}
“\((ivr1, i\text{Vnr1}, i\text{coV11} := \text{Vr1}, \text{Vnr1}, \text{coV11}) \); \(SV ; f\text{Vr}, f\text{Vnr} := \text{Vr}, \text{Vnr}\)
\(; \text{Vr1}, \text{Vnr1}, \text{coV11} := ivr1, i\text{Vnr1}, i\text{coV11})[\text{live} f\text{Vr}, f\text{Vnr}, \text{Vr1}, \text{Vnr1}, \text{coV11}, \text{coV12}] \)

Finally, for big step 3 above, in which we eliminate redundant backup of final values, we observe
“\(((\text{ISV} ; f\text{Vr1}, f\text{Vr2}, f\text{Vr3}, f\text{Vnr1}, f\text{Vnr2}, f\text{Vnr3}) := \text{Vr1}, \text{Vr2}, \text{Vr3}, \text{Vnr1}, \text{Vnr2}, \text{Vnr3}) \)
\(; \text{Vr1}, \text{Vnr1}, \text{coV11} := ivr1, i\text{Vnr1}, i\text{coV11})[\text{Scov} \)
\(; \text{Vr1}, \text{Vr2}, \text{Vr3}, \text{Vnr1}, \text{Vnr2}, \text{Vnr3} := f\text{Vr1}, f\text{Vr2}, f\text{Vr3}, f\text{Vnr1}, f\text{Vnr2}, f\text{Vnr3})[\text{live} \text{Vr}, \text{Vnr}, \text{coV}] \)”
CHAPTER 11. PENLESS SLIDING

= \{\text{split assignment: } (fVr1,fVr2,fVnr1,fVnr2) \circ (Vr3, Vnr3)\}

= \{\text{swap statements: } (fVr3,fVnr3) \circ (Vr1, Vnr1, coV11, iVr1, iVnr1, icoV11) \text{ and } (Vr1, Vnr1, coV11) \circ (fVr3,fVnr3, Vr3, Vnr3)\}

= \{\text{assignment-based sub.: } Vr3 \circ (fVr3 \cup glob.ScoV) \text{ and } fVr3 \circ def.ScoV\}

= \{\text{swap statements: } (fVr3,fVnr3) \circ glob.ScoV[fVr3 \setminus Vr3] \text{ and } def.ScoV[fVr3 \setminus Vr3] \circ (fVr3,fVnr3, Vr3, Vnr3)\}

= \{\text{assignment-based sub.: } (fVr3,fVnr3) \circ (Vr3, Vnr3)\}

= \{\text{assignment-based sub.: } (fVr1,fVr2,fVnr1,fVnr2) \circ (Vr3, Vnr3)\}
\[= \{\text{remove redundant self-assignment;}
\text{remove dead assignment: (fVr3, fVnr3)} \odot (fVr1, fVr2, fVnr1, fVnr2, coV)}\]

\[ (((ISV ; fVr1, fVr2, fVnr1, fVnr2 := Vr1, Vr2, Vnr1, Vnr2 ; Vr1, Vnr1, coV11 := iVr1, iVnr1, icoV11 ; ScoV[Vr3 \setminus Vr3] ; Vr1, Vr2, Vnr1, Vnr2 := fVr1, fVr2, fVnr1, fVnr2)[live Vr, Vnr, coV] \ldots .
\]

For our example above, we end up with

\[
\begin{array}{|c|c|}
\hline
\text{; i, sum := 0, 0} & \text{i, prod := 0, 1} \\
\text{; while i<a.length do} & \text{; while i<a.length do} \\
\text{i, sum :=} & \text{i, prod :=} \\
\text{i+1, sum+ a[i]} & \text{i+1, prod* a[i]} \\
\text{od} & \text{od} \\
\text{; out << sum} & \text{; out << prod} \\
\hline
\end{array}
\]

Note how in the complement, this time, the original variable sum is used where fsum was used before. This was made possible by the successful merging of sum with its two backup variables isum and fsum.

\section{11.2 Compensation-free (or penless) co-slicing}

Since we consider the renaming of variables (by final-use substitution, when co-slicing) as part of sliding's compensation, we accordingly consider a co-slice with no renaming, or with all initial renaming eventually undone, as in the above, a compensation-free co-slice. Since in our metaphor of slides and sliding, compensatory code is written using a non-permanent pen on top of printed transparencies, the merging can be thought of as the erasure of such earlier writing.

Hence, compensation-free co-slices will also be termed penless co-slices. Accordingly, the process of producing such co-slices will be termed penless co-slicing.

We define a penless co-slice to be a co-slice that involve no renaming and thus no compensation, in the following way:
\[\text{penless-co-slice.} S. V. Vr \triangleq (\text{co-slice.} S. V. Vr.fVr)[fVr \setminus Vr]\text{ where}\]
\[fVr := \text{fresh.}(Vr, (V \cup \text{glob.} S)).\]

Note that since normal substitution is defined only when the new names are fresh, a penless co-slice is well-defined only when all reused variables are gone (from the co-slice). That is, \[\text{penless-co-slice.} S. V. Vr\text{ is well-defined when } Vr \odot \text{glob.}(\text{co-slice.} S. V. Vr.fVr).\]
Now that the elimination of redundant backup variables and the construction of penless co-slices have been formalised, we are in position to derive preconditions for \textit{compensation-free sliding}, or \textit{penless sliding}, as in the following.

### 11.3 Sliding with penless co-slices

The following is a sliding transformation with penless co-slicing and with the elimination of redundant backup variables:

\section*{Transformation 11.2.} Let $S$ be any core statement and let $V_r, V_{nr}$ be any two disjoint (user selected) sets of variables to be extracted, with $V_r$ to be made available for reuse in the complement; then

\[
\begin{align*}
\textcolor{white}{\text{S \subseteq}} & \quad \textcolor{white}{\text{S \subseteq}} \\
\{ \text{var } iV_{nr1}, iV_{nr2}, fV_{nr1}, fV_{nr2} \} & \quad \{ \text{var } iV_{nr1}, iV_{nr2}, fV_{nr1}, fV_{nr2} \} \\
; \ iV_{nr1}, iV_{nr2} := V_{nr1}, V_{nr2} & \quad ; \ iV_{nr1}, iV_{nr2} := V_{nr1}, V_{nr2} \\
; \ SV & \quad ; \ Sv \\
; \ fV_{nr1}, fV_{nr2} := V_{nr1}, V_{nr2} & \quad ; \ V_{nr1}, V_{nr2} := V_{nr1}, V_{nr2} \\
\} \quad \} \\
\end{align*}
\]

where

- $V_r', V_{nr}' := (V_r \cap \text{def}.S), (V_{nr} \cap \text{def}.S)$,
- $\text{coV} := \text{def}.S \setminus (V_r', V_{nr}')$,
- $\text{SV} := \text{slice}.S(V_r', V_{nr}')$,
- $\text{ScoV} := \text{penless-co-slice}.S(V_r', V_{nr}')$,
- $V_{nr1}, V_{nr2} := (V_{nr}' \cap \text{input}.\text{ScoV}), (V_{nr}' \cap (\text{def}.\text{ScoV} \setminus \text{input}.\text{ScoV}))$,
- $\text{coV11} := (\text{coV} \cap \text{def}.\text{SV} \cap \text{input}.\text{ScoV})$

and

\[
\begin{align*}
\text{fresh.}((V_{nr1}, V_{nr2}, V_{nr1}, V_{nr2}), ((V_r, V_{nr}) \cup \text{glob}.S)) \\
\end{align*}
\]

provided $V_r' \circ \text{glob}.(\text{co-slice}.S(V_r', V_{nr}'), V_r'.fV_r)$ for any fresh $fV_r$.

\section*{Proof.}

$S$
\[ \subseteq \] (Refinement 11.1 with \( V_r, V_{nr}, ScoV := V_{r'}, V_{nr'}, co-slice.S.(V_{r'}, V_{nr'}).V_{r'}.fV_{r} \) on fresh \( fV_{r} \): \( (V_{r'}, V_{nr'}, coV) = def.S \) by def. of \( V_{r'}, V_{nr'}.coV \); 
\( S[\text{live } V_{r'}, V_{nr'}] \subseteq SV[\text{live } V_{r'}, V_{nr'}] \) by Q1 of slice; 
\( S[\text{final-use } V_{r'} \setminus fV_{r}][\text{live } coV] \subseteq (co-slice.S.(V_{r'}, V_{nr'}).V_{r'}.fV_{r})[\text{live } coV] \) by Q1 of co-slice; 
\( def.SV \subseteq def.S \) by Q2 of slice; similarly 
\( def.ScoV \subseteq def.S \) by Q2 of co-slice; 
\( (iv_{nr1}, ico\ V\ 11, fV_{nr1}, fV_{nr2}) \circ \text{glob}.S \) by Q1 of fresh; 
note that \( V_{r1}, V_{r2}, V_{r3} := 0, 0, V_{r'} \) due to the proviso; 
consequently \( iV_{r1}, iV_{r2}, fV_{r1} \) and \( fV_{r2} \) are all empty; also note (for our \( ScoV \)) penless-co-slice.S.(\( V_{r'}, V_{nr'} \)).\( V_{r'} = (co-slice.S.(V_{r'}, V_{nr'}).V_{r'}.fV_{r})[fV_{r} \setminus V_{r'}] \) by def. of penless-co-slice (which is indeed well-defined due to the proviso))

\[ \text{" } (iv_{nr1}, ico\ V\ 11 := vnr1, co\ V\ 11 ; SV ; fV_{nr1}, fV_{nr2} := vnr1, vnr2 } \]
\[ \text{\quad } V_{nr1}, co\ V\ 11 := iV_{nr1}, ico\ V\ 11 ; ScoV ; V_{nr1}, V_{nr2} := fV_{nr1}, fV_{nr2} \]
\[ \text{\quad } [\text{live } V_{r'}, V_{nr'}.coV] \text{ "} \]

\[ = \text{\{} \text{def. of live: (def.SV } \cup \text{ def.ScoV} \} \subseteq (V_{r'}, V_{nr'}.coV) \text{ (again, Q2 of slice and co-slice) } \}
\[ \text{\quad } \text{\{} (iv_{nr1}, ico\ V\ 11, fV_{nr1}, fV_{nr2}) \circ (vnr1, vnr2, coV) \text{ \}} \]

\[ \text{\" } [\text{var } iV_{nr1}, ico\ V\ 11, fV_{nr1}, fV_{nr2} ; ] \]
\[ \text{\quad } iV_{nr1}, ico\ V\ 11 := vnr1, co\ V\ 11 ; SV ; fV_{nr1}, fV_{nr2} := vnr1, vnr2 \]
\[ \text{\quad } V_{nr1}, co\ V\ 11 := iV_{nr1}, ico\ V\ 11 ; ScoV ; V_{nr1}, V_{nr2} := fV_{nr1}, fV_{nr2} ] \text{ "} \]

11.4 Summary

In this chapter, an approach for reducing compensation after sliding has been developed. Redundant backup variables of initial and final values have been eliminated. This elimination has been conducted by the removal of dead assignments and by merging such backup variables with their original counterparts. When eliminating backup of reusable final values of extracted variables, those had to be removed from the complement. This has been formalised through a concept of compensation-free co-slices, or penless co-slices.

A penless co-slice is constructed by first introducing reusable variables through final-use substitution, then slicing for the remaining variables, and finally undoing the substitution. It is interesting to see that such a co-slice is potentially smaller than the corresponding slice (of non-extracted variables).

A sliding transformation whose complement is a penless co-slice has been developed. We call
it *penless sliding*. Moreover, if all backup variables are successfully eliminated when sliding, as was the case in the given example, we say the result is *completely penless*.

In this light, the KH approach to arbitrary method extraction (both KH00 [38] and KH03 [39], apart from some specific treatment of jumps in the latter) can be described as completely penless. Indeed, the approach taken in the last two chapters, leading to the formulation of penless co-slices and penless sliding has been inspired by their algorithms as well as their criticism of Tuck’s lack of data flow from slice to complement (as stated in [39][37]).

Looking back at the sliding transformations of the current and previous chapters, we note that the user was asked to provide not only the statement in scope $S$ and variables for extraction $V$, as was the case in the earlier sliding transformation (of Chapter 9), but also the subset $V_r$ of extracted variables to be reused in the complement. However, our original formulation of slice extraction (in Definition[1.1]) had no mention of such $V_r$. When the goal is to extract precisely the slice of $S$ on $V$, whilst producing the smallest possible complement, one could ask “which subset $V_r$ would yield the smallest possible complement?” This question, as well as a related question on sliding itself will be treated in the next chapter.
Chapter 12

Optimal Sliding

In previous chapters, all co-slicing related transformations assumed the subset of (extracted) variables to be made reusable, $V_r$, is given. In this chapter, however, that assumption is waived.

This immediately raises a variety of optimisation problems. When extracting the computation of $V$ in $S$, using a certain co-slice related sliding transformation, which partition of def.$S$ into $((V_r, V_{nr}), coV)$ — with $(V_r, V_{nr})$ extracted, and $V_r$ offered for reuse — would yield an optimal result? Surely we need to be more specific in describing what is meant by ‘optimal’, and which transformation is being applied.

In this chapter, we focus on sliding with penless co-slices (i.e. Transformation 11.2 from the preceding chapter). For any given program statement $S$ and set of variables to be extracted $V$, an optimal solution will identify a set of variables $V'$ (possibly larger than $V$ itself, as will be explained shortly), made of subsets $(V_r, V_{nr})$, for which the extracted slice $SV'$ will be precisely $slice.S.V$; its complement, the penless co-slice $ScoV$, must end up being the smallest possible, in terms of the number of individual assignments in it.

It should be noted that we do not mean to consider any substatement in our search: Finding such minimal co-slices is in general impossible, just as finding minimal slices is — it is equivalent to solving the halting problem [64]. Instead, the goal is to find the smallest out of all possible results of our specific (penless) co-slicing algorithm. In our quest, the program statement to be co-sliced shall be given and fixed, whereas the set of variables on which to co-slice, as well as its subset of variables to be made available for reuse, shall vary.

We begin our search for an optimal solution by devising an algorithm to find the smallest possible penless co-slice for given $S$ and $V$. We then complete the solution by observing that the given $V$ itself is not necessarily the best option for the set of extracted variables $V'$. As it turns out, some larger sets may yield precisely the same extracted slice, and with enhanced opportunities for reuse, thus possibly yielding an even smaller complement.
12.1 The minimal penless co-slice

A statement $S$ and set of variables $V$ have at most $2^N$ different penless co-slices (with $N = |V|$). This is so since any subset $Vr$ of $V$ can be offered for reuse, thus possibly leading to a different co-slice. (It should however be remembered that not all subsets necessarily yield well-defined penless co-slices.)

Let the size of a program statement be determined by the number of individual assignments in it. With this definition, is there (for given $S$ and $V$, with different reusable subsets $Vr$) a single smallest result to our algorithm of penless co-slicing? If so, which subset $Vr$ yields it? And how can this $Vr$ be found?

Our conjecture is that indeed there is a single smallest penless co-slice (to any given $S$ and $V$). How do we find it? Surely, one could try all subsets of $V$, composing all possible co-slices and measuring the size of the penless ones. But this algorithm would be very expensive. In terms of time complexity, it would grow exponentially with $|V|$. Is there a faster solution?

12.1.1 A polynomial-time algorithm

The size of (penless or not) co-slices, for any given $S$ and $V$, is anti-monotone with respect to the set $Vr$ of reusable variables. Thus, if $Vr1$ and $Vr$ are two sets which lead to valid penless co-slices (in the context $S, V$) — we refer to such sets as ‘mergeable’ as they can be merged with their corresponding backup variables, after co-slicing — with $Vr1 \subseteq Vr \subseteq V$, we have

$$|\text{penless-co-slice}.S.V.Vr| \leq |\text{penless-co-slice}.S.V.Vr1|.$$  

(Recall the size of a statement here is the number of individual assignments.)

In fact, we should be looking for the largest $Vr$ that yields a penless co-slice. Why largest? Due to anti-monotonicity of the size of penless co-slices (with respect to the set of reusable variables), such a penless co-slice will never be larger than the penless co-slice of any other mergeable set of reusable variables.

Is there only one such largest set? Yes, due to the following definitions and observation.

After co-slicing, the variables in $Vr \setminus \text{glob.(co-slice}.S.V.Vr.fVr)$ are definitely mergeable (i.e. they can be merged with their corresponding members of $fVr$) whereas variables in $Vr \cap \text{glob.(co-slice}.S.V.Vr.fVr)$ are considered non-mergeable. We thus define the set of mergeable reusable variables (after co-slicing of $S, V$ with reusable $Vr \subseteq V$ and fresh $fVr$, i.e. $fVr \diamond (V \cup \text{glob}.S)$) as

$$\text{mergeable}.S.V.Vr \triangleq Vr \setminus \text{glob.(co-slice}.S.V.Vr.fVr).$$  

Accordingly, variables in $Vr \setminus \text{mergeable}.S.V.Vr$ are said to be non-mergeable.

Moreover, when all members of a set of reusable variables $Vr$ are mergeable with respect to $S, V, Vr$ (i.e. when $Vr = \text{mergeable}.S.V.Vr$ which is the case iff $Vr \diamond \text{glob.(co-slice}.S.V.Vr.fVr)$).
we say \( V_r \) is ‘penless’ with respect to \( \text{co-slice}.S.V \).

**Lemma 12.1.** Penlessness is closed under set-union. That is, if \( V_r1 \) and \( V_r2 \) are two penless subsets of \( V \), their union \( V_r1 \cup V_r2 \) is penless too.

**Proof.** Assuming the two subsets \( V_r1 \) and \( V_r2 \) of extracted variables \( V \) are both penless with respect to \( \text{co-slice}.S.V \), we need to show the union \( V_r3 := (V_r1 \cup V_r2) \) is penless too.

On the one hand, we observe

\[
\begin{align*}
\text{glob.}(\text{co-slice}.S.V3.fV_r3) &= \{\text{def. of co-slice; let } coV := \text{def}.S \setminus V\} \\
&= \text{glob.}(\text{slice}.S[\text{final-use } V_r3 \setminus fV_r3].coV) \\
&= \{\text{stepwise final-use sub. (see Section E.3): let } V_r21 := V_r2 \setminus V_r1\} \\
&= \text{glob.}(\text{slice}.S[\text{final-use } V_r1 \setminus fV_r1][\text{final-use } V_r21 \setminus fV_r21].coV) \\
&\subseteq \{\text{Lemma 12.2 see below}\} \\
&= fV_r21 \cup \text{glob.}(\text{slice}.S[\text{final-use } V_r1 \setminus fV_r1].coV) \\
&= \{\text{def. of co-slice}\} \\
&= fV_r21 \cup \text{glob.}(\text{co-slice}.S.V/V_r1.fV_r1) .
\end{align*}
\]

Thus \( V_r1 \circ \text{glob.}(\text{co-slice}.S.V.V3.fV_r3) \) (due to the freshness of \( fV_r21 \) and penlessness of \( V_r1 \) in \( \text{co-slice}.S.V \)). On the other hand, we have

\[
\begin{align*}
\text{glob.}(\text{co-slice}.S.V.V3.fV_r3) &= \{\text{def. of co-slice; let } coV := \text{def}.S \setminus V\} \\
&= \text{glob.}(\text{slice}.S[\text{final-use } V_r3 \setminus fV_r3].coV) \\
&= \{\text{stepwise final-use sub. (see Section E.3): let } V_r11 := V_r1 \setminus V_r2\} \\
&= \text{glob.}(\text{slice}.S[\text{final-use } V_r2 \setminus fV_r2][\text{final-use } V_r11 \setminus fV_r11].coV) \\
&\subseteq \{\text{again Lemma 12.2 see below}\} \\
&= fV_r11 \cup \text{glob.}(\text{slice}.S[\text{final-use } V_r2 \setminus fV_r2].coV) \\
&= \{\text{def. of co-slice}\} \\
&= fV_r11 \cup \text{glob.}(\text{co-slice}.S.V.V2.fV_r2) .
\end{align*}
\]

Thus, similarly to the preceding derivation, \( V_r2 \circ \text{glob.}(\text{co-slice}.S.V.V3.fV_r3) \). We then conclude (from set theory and the definition of \( V_r3 \)) the desired \( V_r3 \circ \text{glob.}(\text{co-slice}.S.V.V3.fV_r3) \).
Lemma 12.2. Let $S, X, fX, Y$ be any core statement and three sets of variables, respectively; then

$$\text{glob}(\text{slice}.S[fX], Y) \subseteq fX \cup \text{glob}(\text{slice}.S.Y)$$

provided $fX \diamond ((X, Y) \cup \text{glob}.S)$.

Proof. Recall the definition of slice. There, the given program statement is first translated into SSA, where it is sliced in a flow-insensitive way, before returning from SSA.

The difference between the SSA versions of $S$ and $S[fX]$ is only in the references to $fX$, since final-use substitution changes only uses, and no definition. So both versions have the same sets of defined variables and the same sets of slides, with potential differences in the used variables on those slides. These potential differences, in turn, may lead to differences in the respective relations of slide dependence. Since $fX \diamond \text{def}.S[fX]$, the introduced uses of $fX$ do not yield any new slide dependence.

Consequently, representing the relations of slide dependence as a set of pairs, the set of slide dependences of the SSA version of $S[fX]$ is a (not necessarily strict) subset of the corresponding set for $S$ itself. Thus, the slide-independent set $YL_{f}^{*}$ (i.e. the reflexive transitive closure of the set $YLF$ of final instances of $Y$) of the former, is a subset of the corresponding set of the latter. The result is that, excluding $fX$ itself, and even after returning from SSA, the set of global variables in the former is a subset of the global variables in the latter.

Finally, how do we find the largest penless set? The following observations suggests an optimistic approach.

Lemma 12.3. Mergeability is monotone with respect to co-slicing (in the set of reusable variables). That is,

$$\text{mergeable}.S.Vr1 \subseteq \text{mergeable}.S.(Vr1, Vr2).$$

Proof.

$$\text{mergeable}.S.Vr1$$

$$= \{\text{def. of mergeable}\}$$

$$Vr1 \setminus \text{glob}.(\text{co-slice}.S.Vr1.fVr1)$$

$$= \{\text{set theory: } fVr1 \circ Vr1\}$$

$$Vr1 \setminus (\text{glob}.(\text{co-slice}.S.Vr1.fVr1) \setminus fVr1)$$

$$\subseteq \{\text{see below}\}$$

$$Vr1 \setminus (\text{glob}.(\text{co-slice}.S.(Vr1, Vr2).(fVr1, fVr2)) \setminus (fVr1, fVr2))$$
Corollary 12.4. When reducing the set of reusable variables from \((Vr1, Vr2)\) to \((Vr1, Vr2)\), when seeking the largest set of penless reusable variables — is the following. When reducing the set of reusable variables from \((Vr1, Vr2)\) to \((Vr1, Vr2)\), when seeking the largest set of penless reusable variables — is the following.

\(\text{(set theory: } (fVr1, fVr2) \odot Vr1)\)

\(Vr1 \\ glob. (\text{co-slice}. S.V. (Vr1, Vr2). (fVr1, fVr2))\)

\(\subseteq\) \text{(set theory)}

\((Vr1, Vr2) \\ glob. (\text{co-slice}. S.V. (Vr1, Vr2). (fVr1, fVr2))\)

\(=\) \text{(def. of mergeable)}

mergeable \(S.V. (Vr1, Vr2)\).

A useful property of co-slicing (for the third step above) is

\((\text{glob}. (\text{co-slice}. S.V. (Vr1, Vr2). (fVr1, fVr2)) \setminus (fVr1, fVr2)) \subseteq (\text{glob}. (\text{co-slice}. S.V. Vr1.fVr1) \setminus fVr1)\).

To see why this is so, we observe

\(\text{glob}. (\text{co-slice}. S.V. (Vr1, Vr2). (fVr1, fVr2)) \setminus (fVr1, fVr2)\)

\(=\) \text{(def. of co-slice; let } coV := def.S \setminus V\)

\(\text{glob}. (\text{slice}. S[\text{final-use } Vr1, Vr2 \setminus fVr1, fVr2, co V] \setminus (fVr1, fVr2)\)

\(=\) \text{(stepwise final-use sub. (see Section E.3))}

\(\text{glob}. (\text{slice}. S[\text{final-use } Vr1 \setminus fVr1][\text{final-use } Vr2 \setminus fVr2, co V] \setminus (fVr1, fVr2)\)

\(=\) \text{(set theory)}

\((\text{glob}. (\text{slice}. S[\text{final-use } Vr1 \setminus fVr1][\text{final-use } Vr2 \setminus fVr2, co V] \setminus (fVr1, fVr2) \setminus fVr1\)

\(\subseteq\) \text{(Lemma 12.2 with } S, X, fX, Y := S[\text{final-use } Vr1 \setminus fVr1], Vr2, fVr2, \text{ co } V\)

\(\text{glob}. (\text{slice}. S[\text{final-use } Vr1 \setminus fVr1], co V \setminus fVr1\)

\(=\) \text{(def. of co-slice; co } V = \text{def}.S \setminus V\)

\(\text{glob}. (\text{co-slice}. S.V. Vr1.fVr1) \setminus fVr1\).

\(\square\)

An interesting consequence of the monotonicity of mergeability — one which calls for an optimistic algorithm, when seeking the largest set of penless reusable variables — is the following.

Corollary 12.4. When reducing the set of reusable variables from \((Vr1, Vr2)\) to \((Vr1, Vr2)\), when \(Vr1 \subseteq Vr2\) and \(Vr1 \odot \text{mergeable}. S.V. (Vr1, Vr2)\), the subset \(Vr1\) of non-mergeable variables in the former remains non-mergeable in the latter (i.e. \(Vr1 \odot \text{mergeable}. S.V. (Vr1, Vr2)\)).

Proof. Due to the monotonicity of mergeability (Lemma 12.3 above), any member of \(Vr1 \cap \text{mergeable}. S.V. (Vr1, Vr2)\) would have to be in \(Vr1 \cap \text{mergeable}. S.V. (Vr1, Vr2)\) as well (due to \(Vr2 \subseteq Vr2\)), thus contradicting the assumption of \(Vr1\) being non-mergeable in \(\text{co-slice}. S.V. (Vr1, Vr2)\).

\(\square\)
Given a core statement $S$ and variables of interest $V$, compute the largest subset $\text{largest-penless-reusable} . S . V$ of $V$ which, when offered for reuse, yields a penless co-slice $(\text{penless-co-slice}. S . (\text{largest-penless-reusable} . S . V))$, which is in turn not larger than any other penless co-slice of $S, V)$, as follows:

$$
$$

$$
\text{largest-penless-reusable-rec} . S . V . Vr \equiv \text{if nonMergeable} = \emptyset \text{ then } Vr \text{ else }
$$

$$
\text{largest-penless-reusable-rec} . S . V . (Vr \setminus \text{nonMergeable}) \text{ fi}
$$

where $\text{nonMergeable} := \text{glob} . (\text{co-slice}. S . V . Vr . fVr) \cap Vr$

and $fVr := \text{fresh} . (Vr, (V \cup \text{glob} . S))$.

Figure 12.1: An algorithm for finding the largest-penless-reusable set.

Remark: the above property should not be confused with ‘non-mergeability is anti-monotone with respect to co-slicing’. In fact, judging by our definition of non-mergeability, the latter is not true. When decreasing the set of reusable variables, say from $(Vr1, Vr2)$ to $Vr1$, a non-mergeable variable from $Vr2$ will no longer be considered either (mergeable or non-mergeable), as it will no longer be offered for reuse.

So with an optimistic approach, the algorithm begins by trying to reuse all extracted variables $V$. It then removes all non-mergeable variables. Now, should we trust the result (i.e. the set $Vr := \text{mergeable} . S . V . V$) to be penless (i.e. $Vr = \text{mergeable} . S . V . Vr$)? Unfortunately that is not necessarily so. By no longer reusing variables in $V \setminus Vr$, the set of global variables $\text{glob} . (\text{co-slice}. S . V . Vr . fVr) \setminus fVr$ is possibly larger than the corresponding $\text{glob} . (\text{co-slice}. S . V . fV) \setminus fV$; the former might include members of $Vr$, thus rendering $Vr$ non-penless. However, such variables can subsequently be removed, repeatedly.

Hence, the algorithm — see Figure 12.1 — is optimistic and recursive. Starting with the largest available set $V$, we repeatedly identify and remove non-mergeable variables, until a fixed point is reached.

12.2 Slice inclusion

When sliding $V$ in $S$, e.g. in Transformation 11.2, the extracted computation consists of the full slice of $V$ in $S$, i.e. $\text{slice}. S . V$. The complement, in turn, consists of the code for computing the remaining results $\text{co} V := \text{def} . S \setminus V$, i.e. $\text{penless-co-slice}. S . V . Vr$, with $Vr$ (being e.g. $\text{largest-penless-reusable} . S . V$, as shown above), a subset of $V$ of extracted variables whose final
extracted value is to be offered for reuse. Variables in $coV$, however, might also be modified in the extracted slice, if those contribute to the computation of $V$. In such a case, the compensatory code ensures (through backup variables) those modifications do not interfere with the eventual computation of $coV$ in the complement.

With this transformation, the final value of the extracted variables can be reused in the complement. But how about the final value of other variables? In the following example, an attempt to slide the computation of $avg$ (and reusing it in the complement), would lead to duplication of the code for computing $sum$.

```
; i,sum,prod := 0,0,1
; while i<a.length do
  i,sum,prod :=
    i+1,sum+a[i],prod*a[i]
  od
; avg := sum/a.length
; out << sum
; out << prod
; out << avg
```

The result will look this way:

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

Notice that the final value of $avg$ was successfully reused. In contrast, the whole computation of $sum$ had to be duplicated. The reason is that its value at the end of the extracted slice was ignored, instead of being offered for reuse through final-use substitution (like $avg$).

In general, there is no reason to restrict final-use substitution to the set of extracted variables, $V$. All other variables whose final value is computed in the extracted slice might be good candidates for reuse too. In the example above, it was $sum$ whose final value was computed both in
CHAPTER 12. OPTIMAL SLIDING

the slice and the complement.

How can we tell it was the final value of a variable that was computed in the extracted slice? This is the case whenever the full slice of such a variable, with respect to the program scope, is included in the extracted code. In general, we can say that all variables whose slices are included in the slice for $V$ are candidates for final-use substitution. We denote this set $V'$ and propose to update our slice-extraction transformations to extract that extended set rather than the requested set $V$.

In the example above, where $V$ was \{avg\}, the corresponding $V'$ set includes \{avg, sum, i\}. Applying Transformation 11.2 to that latter set, with the largest penless reusable set $V_r := \{avg, sum\}$, would therefore lead to:

Note that this time, the final extracted value of sum was reused in the complement, instead of being ignored and thus recomputed. This resulting code is considered better than the previous result in the sense that the code for computing sum is no longer duplicated. And in terms of our optimisation problem, it yields a smaller co-slice, with less assignments.

On the other hand, we now have more compensatory code. For understanding this, we further note that the largest set $V_r$ used for final-use substitution was \{avg, sum\}. The variable $i$ was excluded since intermediate values are used in the complement, for the computation of prod. Instead, the slice for $i$ was duplicated and its modifications in the complement were ignored through a backup variable $fi$. In this case, considering levels of code duplication, ignoring effects on $i$ in the extracted slice, instead, would have been as good. However, in terms of the number of backup variables, the latter would have been better.

Accordingly, when two sliding combinations are similar in terms of code duplication, it might
be desirable to choose the one that minimizes the need for backup variables, as those entail both extra storage and time for copying.

In this thesis, we leave this aspect (of minimizing such compensatory code) alone, and focus solely on levels of code duplication, as displayed by the number of individual assignments in the co-slice.

### 12.3 The optimal sliding transformation

**Transformation 12.5.** Let $S$ be any core statement and $V$ be a set of variables to be extracted; then

\[
\begin{align*}
S & \subseteq \text{"} | \var{iVnr}_1, ico\, V11, fVnr_1, fVnr_2 \text{"} \\
& ; iVnr_1, ico\, V11 := Vnr_1, co\, V11 \\
& ; SV \\
& ; fVnr_1, fVnr_2 := Vnr_1, Vnr_2
\end{align*}
\]

\[
\begin{align*}
Vnr_1, co\, V11 & := iVnr_1, ico\, V11 \\
; ScoV \\
Vnr_1, Vnr_2 & := fVnr_1, fVnr_2 \\
& \text{"}
\end{align*}
\]

where $V'$ is the set of variables in $V \cup \text{def.} S$ whose slice is included in $\text{slice.} S. V$,

- $V_r := \text{largest-penless-reusable.} S. V'$,
- $Vnr := V' \setminus V_r$,
- $co\, V := \text{def.} S \setminus (V_r, Vnr)$,
- $SV := \text{slice.} S. V'$,
- $Sco\, V := \text{penless-co-slice.} S. V'. V_r$,
- $Vnr_1, Vnr_2 := (Vnr \cap \text{input.} Sco\, V), (Vnr \cap (\text{def.} Sco\, V \setminus \text{input.} Sco\, V))$,
- $co\, V11 := (co\, V \cap \text{def.} SV \cap \text{input.} Sco\, V)$

and $(iVnr_1, ico\, V11, fVnr_1, fVnr_2) := \text{fresh.} ((Vnr_1, co\, V11, Vnr_1, Vnr_2), (V \cup \text{glob.} S))$.

**Proof.**

$S$
 CHAPTER 12. OPTIMAL SLIDING

\[ \{ \text{Refinement 11.1 with } V_r, V_{nr}, S_{coV} := V_{r'}, V_{nr'}, \text{co-slice.S.}(V_{r'}, V_{nr'}) \} \]

\[ \text{on fresh } f_{V'r'}: (V_{r'}, V_{nr'}, \text{coV}) = \text{def.S by def. of } V_{r'}, V_{nr'}, \text{coV}; \]
\[ S[\text{live } V_{r'}, V_{nr'}] \subseteq SV[\text{live } V_{r'}, V_{nr'}] \text{ by Q1 of slice; } \]
\[ S[\text{final-use } V_{r'} \setminus f_{V'r'}[\text{live coV}] \subseteq (\text{co-slice.S.}(V_{r'}, V_{nr'}).V_{r'}.f_{V'r})[\text{live coV}] \]

\[ \text{by Q1 of co-slice; } \]
\[ \text{def.SV} \subseteq \text{def.S} \text{ by Q2 of slice; similarly } \]
\[ \text{def.ScoV} \subseteq \text{def.S} \text{ by Q2 of co-slice;} \]
\[ (iV_{nr1}, i\text{coV}11, fV_{nr1}, fV_{nr2}) \circ \text{glob.S} \text{ by Q1 of fresh;} \]
\[ \text{note that } V_{r1}, V_{r2}, V_{r3} := \emptyset, \emptyset, V_{r'} \text{ due to the proviso; } \]
\[ \text{consequently } iV_{r1}, iV_{r2}, f_{V'r1} \text{ and } f_{V'r2} \text{ are all empty; also note (for our } S_{coV}) \]
\[ \text{penless-co-slice.S.}(V_{r'}, V_{nr'}) \cdot V_{r'} = (\text{co-slice.S.}(V_{r'}, V_{nr'}).V_{r'}.f_{V'r})[f_{V'r} \setminus V_{r'}] \]
\[ \text{by def. of penless-co-slice (which is indeed well-defined due to the proviso))} \]

\[ " (iV_{nr1}, i\text{coV}11 := V_{nr1}, \text{coV}11 ; SV ; fV_{nr1}, fV_{nr2} := V_{nr1}, V_{nr2} \]
\[ \cdot V_{r1}, \text{coV}11 := iV_{nr1}, i\text{coV}11 ; S_{coV} ; V_{nr1}, V_{nr2} := fV_{nr1}, fV_{nr2}) \]
\[ [\text{live } V_{r'}, V_{nr'}, \text{coV}] " \]

\[ = \{ \text{def. of live: (def.SV} \cup \text{def.ScoV}) \subseteq (V_{r'}, V_{nr'}, \text{coV}) \text{ (again, Q2 of slice and } \]
\[ \text{co-slice) and } (iV_{nr1}, i\text{coV}11, fV_{nr1}, fV_{nr2}) \circ (V_{r'}, V_{nr'}, \text{coV})} \]

\[ " [\text{var } iV_{nr1}, i\text{coV}11, fV_{nr1}, fV_{nr2} ; \]
\[ iV_{nr1}, i\text{coV}11 := V_{nr1}, \text{coV}11 ; SV ; fV_{nr1}, fV_{nr2} := V_{nr1}, V_{nr2} \]
\[ \cdot V_{r1}, \text{coV}11 := iV_{nr1}, i\text{coV}11 ; S_{coV} ; V_{nr1}, V_{nr2} := fV_{nr1}, fV_{nr2}] " . \]

\[ \square \]

12.4 Summary

This chapter has addressed two related optimisation problems with regards to penless co-slicing and penless sliding, from the preceding chapter. There, the sliding transformation assumed the subset of extracted variables to be reused in the complement is given. Here, in contrast, all possible subsets have been considered, and algorithms for finding the optimal ones have been developed.

The smallest possible penless co-slice is found through an optimistic polynomial time algorithm that assumes all extracted variables should be made reusable, and repeatedly removes those that violate penlessness. When a fixed point is reached, the resulting set is guaranteed to yield the smallest possible penless co-slice. The correctness of this algorithm has been proved through a number of properties of final-use substitution, slicing and co-slicing that have been formally developed.

The optimal sliding transformation, one which extracts precisely the slice of selected variables.
and with the smallest possible penless co-slice (in terms of number of individual assignments), has been shown to involve the extraction of a superset of the selected variables and the smallest penless co-slice, as in our solution to the first optimisation problem. The superset of extracted variables includes the selected set and all other variables whose slice is included in the extracted code.

A relation of slice inclusion, contributing to our detection of optimal sliding, has been introduced by Gallagher and Lyle [22].

In a final note, we return to our declared challenge, from the end of Chapter 2. There, it was shown that if the user requests the extraction of variable \( out \) (or equivalently statements \{1, 2, 4, 6\}), the Tuck transformation would duplicate the entire extracted slice (when not rejecting the extraction), the KH00 algorithm would fail, and the KH03 would insist on extracting statement 3 too, which is illegal in our context of slice extraction.

Our optimal sliding from Transformation 12.5 above, with \( V = \{ out \} \), would detect \( V' = \{ out, sum, i \} \), of which \( Vr = \{ out, sum \} \) would be offered for reuse. Consequently, the challenge of untangling like Tuck whilst minimizing code duplication and improving applicability, like KH03, would be met.

This concludes our current investigation of slice extraction via sliding. Potential applications, for refactoring and otherwise, as well as possible directions for future work will be outlined in the next chapter.
Chapter 13

Conclusion

This thesis has explored the application of program slicing and related analyses to the construction of automatic tools for refactoring.

A theoretical framework for slicing-based refactoring has been developed. The framework has been introduced in Chapter 4 and further extended in Chapter 5 where a new proof method has been developed. The method is based on two complementary types of refinements, i.e. slice-refinement and co-slice-refinement. In our deterministic context, when a program $S'$ is both a slice-refinement and a co-slice-refinement of another program $S$, it is guaranteed to be a full refinement of $S$. This enables the decomposition of a proof, following the specific decomposition applied in a given transformation. The construction of our framework has been finalised in Chapter 8 with the formalisation of a novel program decomposition technique of program slides. We think of a program as represented by a collection of transparency slides. On each such slide, a non-contiguous part of the original program is printed, such that the union of all slides yields back the program itself.

Based on our theoretical framework, a provably correct slicing algorithm has been provided in Chapter 9. The algorithm is based on the observation that slides capture the control flow aspect of slicing, whereas complementary data-flow influences are captured by our binary relation of slide dependence. Thus, a slide-independent set of slides yields a correct slice.

Our framework and slicing algorithm have been applied in solving the problem of slice extraction, as posed in the introduction chapter, via a family of provably correct sliding transformations. Building on existing method-extraction algorithms, our approach shares the advantages of those whilst avoiding some of the respective weaknesses. Thus sliding is successful in providing high levels of accuracy and applicability.

The thesis comes to a conclusion in this chapter, by discussing implications and potential applications of sliding transformations, first in the context of refactoring, and more generally,
later. Furthermore, advanced issues and limitations of sliding are evaluated and some ideas for future work are presented.

13.1 Slicing-based refactoring

13.1.1 Replace Temp with Query

Our journey had started with a promise to offer general automation of Fowler’s refactoring of Replace Temp with Query. This was motivated, in part, by Fowler and Beck’s big refactoring of Convert Procedural Design to Objects and the observation that support for removing temps was missing.

But then, instead, we turned to form and solve the problem of slice extraction (via sliding). The time has come to explain how sliding can contribute to automating Replace Temp with Query. Our observation is that

\[
\text{Replace Temp with Query} = \text{Extract Slice} + \text{Inline Temp} + \text{Merge Temps}
\]

whereby

\[
\text{Extract Slice} = \text{Sliding} + \text{Extract Method}
\]

with Inline Temp a refactoring to eliminate simple temps that are “getting in the way of other refactorings” [20, Page 119], and with Merge Temps as in the elimination of compensatory code after sliding (see e.g. Refinement 11.1).

13.1.2 More refactorings

Our Extract Slice refactoring, automated via sliding, can help with automating some more known (and yet to be supported) refactorings.

Split/merge loops

The Split Loop refactoring is an immediate candidate for automation through sliding. “You have a loop that is doing two things. Duplicate the loop” [69]. Indeed, this is what we did throughout the thesis in most of our examples. However, it should be emphasised that splitting loops is just a special case of our general slice-extraction refactoring.

It should also be remembered that tangled loops are not bad practice as such. It is left to the programmer to apply this refactoring judiciously, e.g. when one of the computations in the loop should be extracted for reuse.

Our separation of refinement and program equivalence rules from actual transformations, throughout the thesis, was made with the understanding that reverse sliding operations, e.g.
for entangling loops (a new Merge Loops refactoring?), may under some circumstances be as desirable. The decision when to apply which refactoring, in our understanding, should be left to the programmer’s good judgement. We merely provide the enabling tools.

**Separate Query from Modifier**

Side effects in functions can be problematic, e.g. hampering potential reuse. “You have a method that returns a value but also changes the state of an object”, is a situation that calls for the Separate Query from Modifier refactoring. “Create two methods, one for the query and one for the modification” is Fowler’s suggestion [20, Page 279].

Cornéllo has formalised this refactoring for simple cases in which the modifier is made of an individual assignment (to an object’s field) and the query returns the old value of the assigned variable (i.e. the field) [11, Page 128]. But what if the querying code is tangled with the modifier? (Indeed this is the case in Fowler’s original example.)

Our observation is that such cases require untangling of non-contiguous code, as is offered by sliding. However, working out exact details of this refactoring will require further investigation. If successful, such application of sliding would yield a novel and advanced solution to this important and highly non-trivial refactoring.

**Arbitrary method extraction**

The Extract Method refactoring is considered so important that its automation, in a behaviour-preserving way, has been declared “The Rubicon” to be crossed by refactoring tools, before those can be considered serious [21]. Furthermore, many of the other catalogued refactoring transformations depend on Extract Method as a building block.

One limitation of method extraction, as formulated and supported to-date, is the insistence on extracting a single fragment of (contiguous) code. Indeed, extracting an arbitrary selection of fragments, from a given program scope, is much harder. We call this generalisation arbitrary method extraction. It involves the extraction of a (not necessarily contiguous) set of program fragments into a single new method.

The most closely related work to sliding, which was introduced early in the thesis (in Section 2.3) and indeed influenced its development, includes the Tucking transformation by Lakhotia and Deprez [40] and two algorithms by Komondoor and Horwitz (KH00 [38] and KH03 [39]). As was mentioned, none of them actually targeted slice extraction, as was defined in Section 1.4.

In fact, it was (different flavours of) arbitrary method extraction that they targeted. Common to all three is that an arbitrary selection of fragments is given as input. They differ (from each other), however, in the rules of the game. Those include (1) the way to determine the enclosing program scope (from which to extract), (2) the applicability conditions (i.e. when to reject a
transformation), (3) which non-selected statements can (or should) be dragged along with the extracted code, (4) what is in the complement and how to compose it with the extracted code, (5) which parts of the program can be duplicated and/or reversed, and (6) what kind of compensation is to be allowed.

Our conjecture is that each of the three arbitrary-method-extraction flavours (i.e. Tuck, KH00 and KH03) can be reduced to slice extraction. The results of an initial investigation in that direction have suggested such reductions indeed exist. Those involve the formulation and reuse of backward slices from internal program points. This can be done with existing sliding transformations, to be performed on the SSA-form rather than the original. Then, the (existing solution’s) applicability conditions should be shown to imply de-SSA-ability of the result. Thus, the existing solutions will be re-formulated, proved correct and automated through sliding.

Consequently, as was the case for slice extraction, corresponding improvements can be expected to present themselves. Those might yield new solutions with higher applicability (compared with Tuck and KH00), higher accuracy of extracted code (Tuck and KH03) and complement (Tuck) reduced levels of duplication and compensation (again, Tuck) and enhanced scalability (mostly the exponential KH00 but also the cubic KH03).

It should be said that this application of sliding is, on the one hand, somewhat surprising (to the author, at least), as it was not at all anticipated (in earlier stages of this research). But on the other hand, it is very reasonable, since the results of Tuck, KH00 and KH03 (in particular the comparison between the former and latter, in [37] and [39]) have directly contributed to the invention of slides and sliding.

13.2 Advanced issues and limitations

In choosing a programming language, we have made some simplifying assumptions, such that formal derivation of the concepts behind sliding has become feasible. It is natural to ask whether sliding transformations can be upgraded to support “real” languages. In what follows, we consider the lifting of some earlier restrictions on the supported language.

Firstly, our assumption of all variables being cloneable was made such that we can make backup of initial and final values, as part of sliding’s compensatory code. Thanks to our penless-sliding effort to remove redundant backup variables, back in Chapter 11 this restriction can be easily lifted. This lifting must be complemented with strengthening sliding’s applicability conditions. Added preconditions will ensure all backup variables of non-cloneable program variables are mergeable and hence removed. Otherwise, the transformation would be rejected. Alternatively, some measures might be taken (as in KH03 where no backup variables are allowed) to avoid the need for such backup.
Secondly, if aliasing is to be permitted, a preliminary step may perform alias analysis before sliding begins. This step would rename variables such that the aliases are seamless to the sliding algorithm. Furthermore, since sliding aims to keep the source as close to the original as possible, this would have to be complemented with a following step to undo the renaming. There, special care would have to be taken with compensatory code, to retrieve backup value of all relevant variables.

Thirdly, allowing structured jumps or even arbitrary control flow, as is the case with existing solutions to arbitrary method extraction, would require a complete reformulation, at least on the lower level of our program analysis and manipulation approach (e.g. laws for propagating assertions). Nevertheless, there appears to be no reason why slides and sliding should not be applicable in such settings. For example, the slide of an assignment would have to include all its control-dependence predecessors instead of merely syntax-tree ancestors. (In our simple language, indeed the latter subsumes the former.) Another example is final-use substitution, whereby instead of propagating assertions as far as possible before making the substitution, it should be possible to formulate the substitution in terms of paths over the control flow graph. There, a final use (of variable \( x \)) is one from which all paths to the exit involve no re-definition (of that \( x \)).

Fourthly, in the presence of exceptions, sliding’s reordering of statements may be problematic in the sense that the transformed program might perform more operations before throwing an exception or might even throw a different exception. When the thrown exceptions are part of the behaviour to be preserved, such reordering should be limited or even completely abandoned. Instead, it should be possible to adopt alternative extraction strategies which involve no reordering of statements. We have proposed two such alternative strategies, one object oriented, and the other aspect oriented [36], in a paper titled “Untangling: A Slice Extraction Refactoring” [17]. Explained in our context of slips and slides, both strategies involve the sliding of slips, without their controlling guards. Such slip sliding would extract the statement of a slip into a method of its own, in the object oriented case, or into an advice in an aspect-oriented context. In the former, the slip would be replaced with a call to the new method whereas in the latter, the extracted advice slip will be associated with a pointcut designator, ensuring it is slid back (i.e. woven) into its original point, before execution.

Fifthly, in the presence of concurrency it is unclear whether sliding is at all appropriate, again, due to reordering of computations. Nevertheless, some solutions to slicing are available for such settings (e.g. [35] [15]). It is possible that as it was for aliasing, new preconditions to sliding may be formed to ensure behaviour preservation.

Finally, supporting procedures, parameters, overloading, and object-oriented constructs (e.g. inheritance and polymorphism) would be a great challenge. Indeed, slicing research has already proposed solutions to such problems, on the one hand, whereas predicate transformers (e.g. for
the language ROOL \cite{11} have been defined and even applied to refactoring, on the other. It is currently unknown whether such advanced solutions will be amenable for supporting formulations of sliding.

In the context of PDG-based slicing, extra language features (e.g. arbitrary control flow \cite{6}) are typically handled by the addition of more edges to the graph. This way, the slicing algorithm itself is oblivious to those features and remains simple. Similarly, it can be expected that sliding, being based on slides and slicing, be enhanced by the addition of slide dependences.

**13.3 Future work**

Sliding, as presented in this thesis, offers an abundance of possible directions for future work. Earlier in this chapter, we have already mentioned a number of possible applications of sliding in implementing known refactorings and in extending method extraction to support arbitrary selections of non-contiguous code fragments.

In our discussion on supporting advanced language features and limitations of sliding, some further ideas have been highlighted, including the support for different strategies of extraction, as in our paper on refactoring into aspects \cite{17}.

Some further ideas may involve the theory behind sliding, or practicalities, or even other applications, beyond the initial domain of refactoring.

**13.3.1 Formal program re-design**

In this thesis, we have restricted our supported language for deterministic constructs. If the earlier section considered the lifting of language restrictions when supporting “real” languages, here we turn the other way, considering the effects of supporting non-determinism and specifications.

Our problem with non-determinism has been related to the duplication of such constructs. As in the above section, this problem can be treated by adding a precondition to sliding, ensuring no non-deterministic choice is duplicated. Alternatively, a mechanism to ensure exact repetition of non-deterministic choices, wherever duplicated, can be installed. The details of such mechanisms would require further work.

It is hoped that with robust support for change of programs and specifications — and sliding may offer a step towards such support — formal methods of program design, and hence of re-design, would become more agile and perhaps, consequently, more widely used.
13.3.2 Further applications of sliding: beyond refactoring

The sliding family of program equivalence and refinement rules, as introduced in the thesis, has been applied to behaviour-preserving transformations for refactoring, with the aim of supporting change in software. Nonetheless, this does not have to stop there. Sliding carries the potential of being relevant and applicable anywhere program equivalence or behaviour-preserving program changes are.

**Software obfuscation**

The sliding refinement rules of Chapters 10 and 11 provide a large universe of equivalent programs, as was explored in the optimisation problems of Chapter 12. In construction, those only differ in the subset of reusable variables and hence in the size and shape of the co-slice. In obfuscation [13], a program is being transformed with the aim of becoming less readable. This is desirable e.g. for software security and protection. In a way, obfuscation is the opposite of refactoring, but as it also involves behaviour-preserving transformations, it may benefit from sliding. Moreover, the large number of equivalent programs carries the potential of rendering the reversal of sliding-based obfuscation harder.

**Clone elimination**

The arbitrary-method-extraction algorithms, by Komondoor and Horwitz [38, 39, 37], target the elimination of clones, or duplicated (not-necessarily contiguous) statements, in existing programs. Their approach eliminates pair of clones as well as clone groups. Since, with some more work, sliding is expected to be made applicable for such method-extraction techniques (as KH00 and KH03), it should also be useful in that context.

**Integration of program variants**

In general, as said, sliding can be expected to be useful wherever program equivalence is. One interesting application of such equivalence is in the integration of variants of a program. This is useful, for example, when a group of programmers is working simultaneously on a given code base. Horwitz, Prins and Reps [31, 32] have suggested some PDG-based and slicing related algorithms of program merging for integration. Those were based on the observation that if two programs are represented by isomorphic dependence graphs, they are equivalent. However, the reverse is not true, obviously, as the problem of program equivalence is in general undecidable. With sliding, we identify a range of equivalent programs whose dependence graphs will not be isomorphic. This is due to duplication of guards and assignments. This sliding-related family of equivalent programs, might, in turn, enhance the capabilities of such program integration algorithms.
CHAPTER 13. CONCLUSION

Optimising compilers

In this thesis, we have adopted some program analyses, representations and transformations from the world of optimising compilers, such as reaching definitions, SSA form, live variables analysis and the related dead-assignments-elimination.

In turn, it should be interesting to investigate the relevance of sliding transformations to that domain. It appears that sliding offers more powerful code-motion transformations than the state of the art.

Programming education

On a different level altogether, it is hoped that slides and sliding, either as a metaphor or in theory and practice, can find their way into the programming education curriculum, especially in the education of non-mathematically inclined programmers. For example, teaching and learning the concept of recursion, with the slideshow metaphor, having a single slide for each iteration, on which values of parameters and local variables are written with an erasable pen, may prove simpler, more tangible than present methods. Furthermore, since often programmers think of programs as slices of non-contiguous code rather than trees or flow graphs, as Weiser has shown [62], it is hoped that representing programs as collections of slides would appeal to programmers.

Beyond programming

Finally, it should be hard but interesting to examine the application of slides and sliding beyond the world of programming. In general, any context of evolvable structured documents could benefit from such techniques.

For example, this thesis has been written, or rather developed, using \LaTeX. Moreover, its content and structure have evolved throughout. Could slicing, slides and sliding not assist in such activities?
Appendix A

Formal Language Definition

This appendix gives a full definition of the language used in this thesis. Each language construct is
given semantics formulated as a \( wp \) predicate transformer. Then, some syntactic approximations
to required semantic properties of that construct, regarding program variables, is given. Finally,
for each construct, the basic requirements (RE1-RE5) are proved. Those were defined back in
Chapter \[4] and are re-stated next. For any statement \( S \) we require

<table>
<thead>
<tr>
<th>RE</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE1</td>
<td>( wp.S ) is universally disjunctive</td>
</tr>
<tr>
<td>RE2</td>
<td>( glob.(wp.S.P) \subseteq ((glob.P \setminus ddef.S) \cup input.S) ) for all ( P )</td>
</tr>
<tr>
<td>RE3</td>
<td>( [wp.S.P \equiv P \land wp.S.true] ) for all ( P ) with ( glob.P \circ ddef.S )</td>
</tr>
<tr>
<td>RE4</td>
<td>( ddef.S \subseteq def.S )</td>
</tr>
<tr>
<td>RE5</td>
<td>( glob.S = def.S \cup input.S )</td>
</tr>
</tbody>
</table>

A.1 Core language

Assignment

\[ wp." X := E \" .P \equiv P[X \setminus E] \] for all \( P \) ;
\[ def." X := E \" \equiv X ; \]
\[ ddef." X := E \" \equiv X ; \]
\[ input." X := E \" \equiv glob.E ; \]
\[ glob." X := E \" \equiv X \cup glob.E . \]

\textbf{RE2:} \( glob.(wp." X := E \" .P) \subseteq (glob.P \setminus ddef." X := E \") \cup input." X := E \"

\[ glob.(wp." X := E \" .P) \]
\[ = \{ wp \text{ of } ";=\}' \]
glob.\(P[X \setminus E]\)
\[
\subseteq \{\text{glob of normal sub.}\}
\]
\((\text{glob.} P \setminus X) \cup \text{glob.} E\)
\[
= \{d\text{def and input of ‘:=’}\}
\]
\((\text{glob.} P \setminus d\text{def.} “ X := E ”) \cup \text{input.} “ X := E ” \)

**RE3:** \(\text{wp.} “ X := E ”. P \equiv P \land \text{wp.} “ X := E ”. \text{true} \) provided \(\text{def.} “ X := E ” \odot \text{glob.} P\)

\[P \land \text{wp.} “ X := E ”. \text{true}\]
\[
= \{\text{wp of ‘:=’}\}
\]
\[P \land \text{true}[X \setminus E]\]
\[
= \{X \odot \text{glob.} \text{true}\}
\]
\[P \land \text{true}\]
\[
= \{\text{identity element of } \land\}\]
\[P\]
\[
= \{\text{redundant normal sub.: } X \odot \text{glob.} P \text{ due to proviso and definition of } \text{def}\}\]
\[P[X \setminus E]\]
\[
= \{\text{wp of ‘:=’}\}
\]
\[\text{wp.} “ X := E ”. P\]

**RE4:** \(d\text{def.} “ X := E ” \subseteq \text{def.} “ X := E ”\): Trivially so, since \(X \subseteq X\).

**RE5:** \(\text{glob.} “ X := E ” = \text{def.} “ X := E ” \cup \text{input.} “ X := E ”\)

\[\text{glob.} “ X := E ”\]
\[
= \{\text{glob of ‘:=’}\}
\]
\[X \cup \text{glob.} E\]
\[
= \{\text{def and input of ‘:=’}\}
\]
\[\text{def.} “ X := E ” \cup \text{input.} “ X := E ”\]
Sequential composition

\[ wp." S1 ; S2 ".P \equiv wp.S1.(wp.S2.P) \] for all \( P \);
\[ def." S1 ; S2 " \equiv def.S1 \cup def.S2 \]
\[ ddef." S1 ; S2 " \equiv ddef.S1 \cup ddef.S2 \]
\[ input." S1 ; S2 " \equiv input.S1 \cup (input.S2 \setminus ddef.S1) \] and
\[ glob." S1 ; S2 " \equiv glob.(S1,S2) \]

\[ \text{RE2: glob.(wp." S1 ; S2 ".P) } \subseteq (\text{glob.} \setminus \text{ddef." S1 ; S2 "}) \cup \text{input." S1 ; S2 " provided glob.(wp.S1.Q) } \subseteq (\text{glob.Q } \setminus \text{ddef.S1}) \cup \text{input.S1 and glob.(wp.S2.P) } \subseteq (\text{glob.P } \setminus \text{ddef.S2}) \cup \text{input.S2 for any predicates P, Q} \]

\[ \begin{align*}
glob.(wp." S1 ; S2 ".P) &= \{ \text{wp of } \circ \} \\
glob.(wp.S1.(wp.S2.P)) \subseteq \{ \text{proviso with } Q := wp.S2.P \} \\
(\text{glob.(wp.S2.P) } \setminus \text{ddef.S1}) \cup \text{input.S1} \subseteq \{ \text{proviso} \} \\
((\text{glob.P } \setminus \text{ddef.S2}) \cup \text{input.S2}) \setminus \text{ddef.S1} \cup \text{input.S1} &= \{ \text{set theory: } (a \cup b) \setminus c = (a \setminus c) \cup (b \setminus c) \text{ and} \\
(d \setminus e) \setminus f = d \setminus (e \cup f) \}
\end{align*} \]

\[ \begin{align*}
(\text{glob.P } \setminus (\text{ddef.S1 } \cup \text{ddef.S2})) \cup (\text{input.S2 } \setminus \text{ddef.S1}) \cup \text{input.S1} &= \{ \text{ddef and input of } \circ \} \\
(\text{glob.P } \setminus \text{ddef." S1 ; S2 ") } \cup \text{input." S1 ; S2 "} \]
\]

\[ \square \]

\[ \text{RE3: } [wp." S1 ; S2 ".P \equiv P \land wp." S1 ; S2 ".true] \text{ provided def." S1 ; S2 " } \circ \text{glob.P,} \\
[wp.S1.Q \equiv Q \land wp.S1.true] \text{ for any } Q \text{ with def.S1 } \circ \text{glob.Q and [wp.S2.R } \equiv R \land wp.S2.true] \text{ for any } R \text{ with def.S2 } \circ \text{glob.R} \]

\[ \begin{align*}
wp." S1 ; S2 ".P &= \{ \text{wp of } \circ \} \\
wp.S1.(wp.S2.P) &= \{ \text{proviso: def.S2 } \circ \text{glob.P} \}
\end{align*} \]

\[ wp.S1.(P \land wp.S2.true) \]
\[ \{ \text{wp}.S_1 \text{ is finitely conjunctive} \} \]
\[ \begin{align*}
\text{wp}.S_1.P & \land \text{wp}.S_1.(\text{wp}.S_2.\text{true}) \\
= & \quad \{ \text{proviso: } \text{def}.S_1 \odot \text{glob}.P \} \\
P & \land \text{wp}.S_1.\text{true} \land \text{wp}.S_1.(\text{wp}.S_2.\text{true}) \\
= & \quad \{ \text{finite conjunctive} \} \\
P & \land \text{wp}.S_1.(\text{true} \land \text{wp}.S_2.\text{true}) \\
= & \quad \{ \text{wp of } \\ ; \\ \text{true} \} \\
P & \land \text{wp}.“\text{ S}_1 ; \text{ S}_2 ”.\text{true} \quad \square
\]

**RE4:** \text{ddef}.“\ S_1 ; \ S_2 ” \subseteq \text{def}.“\ S_1 ; \ S_2 ” \text{ provided } \text{ddef}.S_1 \subseteq \text{def}.S_1 \text{ and } \text{ddef}.S_2 \subseteq \text{def}.S_2: \text{ Indeed } \text{ddef}.S_1 \cup \text{ddef}.S_2 \subseteq \text{def}.S_1 \cup \text{def}.S_2, \text{ due to the proviso and set theory.}

**RE5:** \text{glob}.“\ S_1 ; \ S_2 ” = \text{def}.“\ S_1 ; \ S_2 ” \cup \text{input}.“\ S_1 ; \ S_2 ” \text{ provided } \text{glob}.S_1 = \text{def}.S_1 \cup \text{input}.S_1 \text{ and } \text{glob}.S_2 = \text{def}.S_2 \cup \text{input}.S_2

\[ \begin{align*}
\text{def}.“\ S_1 ; \ S_2 ” & \cup \text{input}.“\ S_1 ; \ S_2 ” \\
= & \quad \{ \text{def and input of } \\ ; \\ \text{true} \} \\
\text{def}.S_1 \cup \text{def}.S_2 \cup \text{input}.S_1 \cup (\text{input}.S_2 \setminus \text{ddef}.S_1) \\
= & \quad \{ \text{set theory: } \text{ddef}.S_1 \subseteq \text{def}.S_1 \} \\
\text{def}.S_1 \cup \text{def}.S_2 \cup \text{input}.S_1 \cup \text{input}.S_2 \\
= & \quad \{ \text{proviso} \} \\
\text{glob}.S_1 \cup \text{glob}.S_2 \\
= & \quad \{ \text{glob of } \\ ; \\ \text{true} \} \\
\text{glob}.“\ S_1 ; \ S_2 ” \quad \square
APPENDIX A. FORMAL LANGUAGE DEFINITION

Alternative construct

\[ wp.IF.P \equiv (B \Rightarrow wp.S1.P) \land (\neg B \Rightarrow wp.S2.P) \] for all \( P \);

\[ def.IF \triangleq def.S1 \cup def.S2 \];

\[ ddef.IF \triangleq ddef.S1 \cap ddef.S2 \];

\[ input.IF \triangleq glob.B \cup input.S1 \cup input.S2 \]; and

\[ glob.IF \triangleq glob.B \cup glob.S1 \cup glob.S2 \].

RE2: \( glob.(wp.IF.P) \subseteq (glob.P \setminus ddef.IF) \cup input.IF \) provided \( glob.(wp.S1.P) \subseteq (glob.P \setminus ddef.S1) \cup input.S1 \) and \( glob.(wp.S2.P) \subseteq (glob.P \setminus ddef.S2) \cup input.S2 \)

\[
glob.(wp.IF.P) = \{ wp \text{ of IF} \}
\]

\[
glob.(B \Rightarrow wp.S1.P) \land (\neg B \Rightarrow wp.S2.P)
\]

\[
= \{ \text{def. of glob; } glob.B = glob.(\neg B) \}
\]

\[
glob.B \cup glob.(wp.S1.P) \cup glob.(wp.S2.P)
\]

\[
\subseteq \{ \text{proviso, twice} \}
\]

\[
glob.B \cup (glob.P \setminus ddef.S1) \cup input.S1 \cup (glob.P \setminus ddef.S2) \cup input.S2
\]

\[
= \{ \text{set theory} \}
\]

\[
(glob.P \setminus (ddef.S1 \cap ddef.S2)) \cup glob.B \cup input.S1 \cup input.S2
\]

\[
= \{ \text{ddef and input of IF} \}
\]

\[
(glob.P \setminus ddef.IF) \cup input.IF \]

RE3: \( wp.IF.P \equiv P \land wp.IF.true \) provided \( def.IF \circ glob.P \)

\[
wp.IF.P
\]

\[
= \{ wp \text{ of IF} \}
\]

\[
(B \Rightarrow wp.S1.P) \land (\neg B \Rightarrow wp.S2.P)
\]

\[
= \{ \text{proviso, twice: } def.IF \circ glob.P \Rightarrow def.S1 \circ glob.P \text{ and } def.S2 \circ glob.P \}
\]

\[
(B \Rightarrow P \land wp.S1.true) \land (\neg B \Rightarrow P \land wp.S2.true)
\]

\[
= \{ \text{dist. of } \Rightarrow \text{ over } \land, \text{ twice} \}
\]

\[
(B \Rightarrow P) \land (B \Rightarrow wp.S1.true) \land (\neg B \Rightarrow P) \land (\neg B \Rightarrow wp.S2.true)
\]

\[
= \{ \text{pred. calc.: } [(Y \Rightarrow Z) \land (\neg Y \Rightarrow Z) \equiv Z] \}
\]

\[
P \land (B \Rightarrow wp.S1.true) \land (\neg B \Rightarrow wp.S2.true)
\]

\[
= \{ wp \text{ of IF} \} \]

\[ P \land \text{wp.IF.true} \]

**RE4:** \( \text{ddef.IF} \subseteq \text{def.IF} \) provided \( \text{ddef.S1} \subseteq \text{def.S1} \) and \( \text{ddef.S2} \subseteq \text{def.S2} \): Indeed \( \text{ddef.S1} \cap \text{ddef.S2} \subseteq \text{def.S1} \cup \text{def.S2} \), due to the proviso and set theory \((a \cap b \subseteq a \cup b)\).

**RE5:** \( \text{glob.IF} = \text{def.IF} \cup \text{input.IF} \) provided \( \text{glob.S1} = \text{def.S1} \cup \text{input.S1} \) and \( \text{glob.S2} = \text{def.S2} \cup \text{input.S2} \)

\[
\begin{align*}
\text{def.IF} \cup \text{input.IF} &\quad = \quad \{ \text{def and input of IF} \} \\
\text{def.S1} \cup \text{def.S2} \cup \text{glob.B} \cup \text{input.S1} \cup \text{input.S2} &\quad = \quad \{ \text{proviso} \} \\
\text{glob.B} \cup \text{glob.S1} \cup \text{glob.S2} &\quad = \quad \{ \text{glob of IF} \} \\
\text{glob.B} &\quad \cup \text{glob.B} \cup \text{input.S} \quad \& \quad \text{glob.IF} \quad \square
\end{align*}
\]

**Repetitive construct**

\([\text{wp.DO.P} \equiv (\exists i : 0 \leq i : (k^i.\text{false}))]\) for all \( P \),

with \( k \) given by (DS:9.44) [13]: \([k.Q \equiv (B \lor P) \land (\neg B \lor \text{wp.S}.Q)]\) ;

\( \text{def.DO} \triangleq \text{def.S} \quad ; \)

\( \text{ddef.DO} \triangleq \emptyset \quad ; \)

\( \text{input.DO} \triangleq \text{glob.B} \cup \text{input.S} \quad ; \) and

\( \text{glob.DO} \triangleq \text{glob.B} \cup \text{glob.S} \quad . \)

**RE2:** \( \text{glob.}(\text{wp.DO.P}) \subseteq (\text{glob.P}\text{\textbackslash\text{ddef.DO}})\cup\text{input.DO} \) provided \( \text{glob.}(\text{wp.S}.Q) \subseteq (\text{glob.Q}\text{\textbackslash\text{ddef.S}})\cup\text{input.S} \)

Recall that \( \text{wp.DO.P} \) is equivalent to \((\exists i : 0 \leq i : (k^i.\text{false}))\) with \([k.Q \equiv (B \lor P) \land (\neg B \lor \text{wp.S}.Q)]\) and that \( \text{ddef.DO} \triangleq \emptyset \) and \( \text{input.DO} \triangleq \text{glob.B} \cup \text{input.S} \). Observing that \( \text{glob.false} \subseteq \text{glob.P} \cup \text{glob.B} \cup \text{input.S} \) is trivially true, we are left to prove

\( \text{glob.}((B \lor P) \land (\neg B \lor \text{wp.S}.Q)) \subseteq \text{glob.P} \cup \text{glob.B} \cup \text{input.S} \) for any \( Q \) with \( \text{glob.Q} \subseteq \text{glob.P} \cup \text{glob.B} \cup \text{input.S} \):

\[ \text{glob.}((B \lor P) \land (\neg B \lor \text{wp.S}.Y)) \]
APPENDIX A. FORMAL LANGUAGE DEFINITION

RE3: \([\text{wp.DO.P} \equiv P \land \text{wp.DO.true}]\) provided \(\text{def.DO} \circ \text{glob.P}\) and \([\text{wp.S.P} \equiv P \land \text{wp.S.true}]\)

Here, recall that \(\text{wp.DO.P}\) is equivalent to \((\exists i : 0 \leq i : k^i.\text{false})\) with \([k.Q \equiv (B \lor P) \land (\neg B \lor \text{wp.S.Q})]\) and \(\text{def.DO} \triangleq \text{def.S}\). Furthermore, note that \(\text{wp.DO.true}\) is equivalent to \((\exists i : 0 \leq i : l^i.\text{false})\) with \([l.Q \equiv \neg B \lor \text{wp.S.Q}]\) due to \text{true} being the zero element of \(\lor\) as well as the identity element of \(\land\). Hence, we need to prove:

\[
[(\exists i : 0 \leq i : k^i.\text{false}) \equiv X \land (\exists i : 0 \leq i : l^i.\text{false})] \tag{A.1}
\]

and we do so by proving (by induction) for all \(j \geq 0\):

\[
[(\exists i : 0 \leq i \leq j : k^i.\text{false}) \equiv P \land (\exists i : 0 \leq i : l^i.\text{false})]
\]

again, provided \(\text{def.DO} \circ \text{glob.P}\) and \([\text{wp.S.P} \equiv P \land \text{wp.S.true}]\):

**Base case:** \(j = 0\)

\[
P \land (\exists i : 0 \leq i \leq 0 : t^i.\text{false})
\]

\[
= \{\text{one point rule}\}
\]

\[
P \land (\exists i : 0 \leq i \leq 0 : t^i.\text{false})
\]

\[
= \{\text{one point rule}\}
\]

\[
P \land t^0.\text{false}
\]

\[
= \{\text{definition of function iteration}\}
\]

\[
P \land \text{false}
\]

\[
= \{\text{zero element of } \land\}
\]
false

\( k^0 \cdot \text{false} \)

\( \{ \text{definition of function iteration} \} \)

\( k^0 \cdot \text{false} \)

\( \{ \text{one point rule} \} \)

\((\exists i : 0 \leq i \leq 0 : k^i \cdot \text{false})\)

**Step:** \( j + 1 \) (with \( 0 \leq j \))

\((\exists i : 0 \leq i \leq j + 1 : k^i \cdot \text{false})\)

\( \{ \text{splitting the range} \} \)

\((\exists i : (0 = i) \lor (1 \leq i \leq j + 1) : k^i \cdot \text{false})\)

\( \{ \text{one point rule and transforming the dummy} \} \)

\( k^0 \cdot \text{false} \lor (\exists i : 0 \leq i \leq j : k^i \cdot \text{false}) \)

\( \{ \text{def. of func. it., twice} \} \)

\( k \cdot (\exists i : 0 \leq i \leq j : k^i \cdot \text{false}) \)

\( \{ \text{def. of } k \} \)

\( (B \lor P) \land (\neg B \lor \text{wp}.S.((\exists i : 0 \leq i \leq j : k^i \cdot \text{false})) \)

\( \{ \text{ind. hypo.} \} \)

\( (B \lor P) \land (\neg B \lor \text{wp}.S.(P \land \exists i : 0 \leq i \leq j : l^i \cdot \text{false})) \)

\( \{ \text{wp}.S \text{ is fin. conj. (and even univ. so)} \} \)

\( (B \lor P) \land (\neg B \lor (\text{wp}.S.P \land \text{wp}.S.((\exists i : 0 \leq i \leq j : l^i \cdot \text{false}))) \)

\( \{ \text{proviso} \} \)

\( (B \lor P) \land (\neg B \lor (P \land \text{wp}.S.\text{true} \land \text{wp}.S.(\exists i : 0 \leq i \leq j : l^i \cdot \text{false})) \)

\( \{ \text{wp}.S \text{ is fin. conj. (and even univ. so)} \} \)

\( (B \lor P) \land (\neg B \lor (P \land \text{wp}.S.(\text{true} \land (\exists i : 0 \leq i \leq j : l^i \cdot \text{false}))) \)

\( \{ \text{id. elem. of } \land \} \)

\( (B \lor P) \land (\neg B \lor (P \land \text{wp}.S.(\exists i : 0 \leq i \leq j : l^i \cdot \text{false}))) \)

\( \{ \lor \text{ distributes over } \land \} \)

\( (B \lor P) \land (\neg B \lor P) \land (\neg B \lor \text{wp}.S.(\exists i : 0 \leq i \leq j : l^i \cdot \text{false}) \)

\( \{ \text{pred. calc.: } [(C \lor D) \land (\neg C \lor D) \equiv D] \} \)

\( P \land (\neg B \lor \text{wp}.S.(\exists i : 0 \leq i \leq j : l^i \cdot \text{false}) \)
= \{\text{def. of } l\}
= P \land l.(\exists i : 0 \leq i \leq j : l^i \cdot \text{false})
= \{\text{id. elem. of } \lor; \text{ } l \text{ is pos. disj. (and even universally so)}\}
= P \land (\text{false} \lor (\exists i : 0 \leq i \leq j : l^i \cdot \text{false}))
= \{\text{def. of func. it., twice}\}
= P \land (l^0 \cdot \text{false} \lor (\exists i : 0 \leq i \leq j : l^{i+1} \cdot \text{false}))
= \{\text{one point rule and transforming the dummy}\}
= P \land (\exists i : (0 = i) \lor (1 \leq i \leq j + 1) : l^i \cdot \text{false})
= \{\text{splitting the range}\}
= P \land (\exists i : 0 \leq i \leq j + 1 : l^i \cdot \text{false})
= \{\land \text{ distributes over } \exists \} (3.11)
= (\exists i : 0 \leq i \leq j + 1 : X \land l^i \cdot \text{false})

RE4: \text{dddf. } \text{DO} \subseteq \text{dddf. } \text{DO} \text{ provided dddf. } \text{S} \subseteq \text{dddf. } \text{S}: \text{ Trivially so since dddf. } \text{DO} \triangleq \emptyset.  

RE5: \text{glob. } \text{DO} = \text{dddf. } \text{DO} \cup \text{input. } \text{DO} \text{ provided glob. } \text{S} = \text{dddf. } \text{S} \cup \text{input. } \text{S}

= \{\text{def and input of DO}\}
= \{\text{proviso}\}
= \{\text{glob of DO}\}

This completes our subset of Dijkstra and Scholten’s guarded commands [13]. The following constructs are extensions borrowed from Morgan [45], with some adaptations as our context requires. Since those constructs were not present in [13], we shall also be responsible for proving requirement \text{RE1} (i.e. universal disjunctivity).
A.2 Extended language

Assertions

\[ \text{wp.}\{B\} ".P \equiv B \land P \] for all \( P \);
\[ \text{def.}\{B\} " \triangleq \emptyset \];
\[ \text{ddef.}\{B\} " \triangleq \emptyset \];
\[ \text{input.}\{B\} " \triangleq \text{glob} \cdot B \]; and
\[ \text{glob.}\{B\} " \equiv \text{glob} \cdot B \].

RE1: \( \text{wp.}\{B\} " \) is universally disjunctive

\[
\begin{align*}
\text{wp.}\{B\} "&(\exists P : P \in Ps : P)
\quad = \quad \{\text{wp of assertions}\}
\quad B \land \exists P : P \in Ps : P
\quad = \quad \{\land \text{distributes over } \exists (3.11)\}
\quad \exists P : P \in Ps : B \land P
\quad = \quad \{\text{again, wp of assertions}\}
\quad \exists P : P \in Ps : \text{wp.}\{B\} ".P
\end{align*}
\]

RE2: \( \text{glob.}(\text{wp.}\{B\} ".P) \subseteq (\text{glob.}P \setminus \text{ddef.}\{B\} " ) \cup \text{input.}\{B\} " \)

\[
\begin{align*}
\text{glob.}(\text{wp.}\{B\} ".P)
\quad = \quad \{\text{wp of assertions}\}
\quad \text{glob.(}B \land P)
\quad = \quad \{\text{def. of glob}\}
\quad \text{glob} \cdot B \cup \text{glob} \cdot P
\quad = \quad \{\text{set theory and ddef and input of assertions}\}
\quad (\text{glob} \cdot P \setminus \text{ddef.}\{B\} " ) \cup \text{input.}\{B\} "
\end{align*}
\]

RE3: \( \text{wp.}\{B\} ".P \equiv P \land \text{wp.}\{B\} " .\text{true} \) provided \( \text{ddef.}\{B\} " \circ \text{glob} \cdot P \)

\[
\begin{align*}
P \land \text{wp.}\{B\} " .\text{true}
\end{align*}
\]
APPENDIX A. FORMAL LANGUAGE DEFINITION

\[
\begin{align*}
&= \{ \text{wp of assertions} \} \\
&P \land B \land \text{true} \\
&= \{ \text{identity element of } \land \} \\
&P \land B \\
&= \{ \text{wp of assertions} \}
\]

\[
\text{wp.}^\times \{ B \}.P
\]

\text{RE4:} \ ddef.\{ B \} \subseteq \text{def.}\{ B \} : \text{Trivially so, since } ddef.\{ B \} = \text{def.}\{ B \} = \emptyset.

\text{RE5:} \ glob.\{ B \} = \text{def.}\{ B \} \cup \text{input.}\{ B \}

\[
\begin{align*}
&= \{ \text{glob of assertions} \} \\
&= \{ \text{def and input of assertions} \} \\
&= \text{def.}\{ B \} \cup \text{input.}\{ B \}
\end{align*}
\]

Local variables

\[
\begin{align*}
&= \{ \text{wp of assertions} \} \\
&P \land B \land \text{true} \\
&= \{ \text{identity element of } \land \} \\
&P \land B \\
&= \{ \text{wp of assertions} \}
\]

\[
\text{wp.}^\times \{ B \}.P
\]

\text{RE1:} \ wp.\{ [\text{var } L \ ; S] \} \text{ is} \text{ universally disjunctive, provided wp.} S \text{ is}

\[
\begin{align*}
&= \{ \text{wp of locals: } L \circ \text{glob.Ps} \}
\end{align*}
\]
\[ wp.S.(\exists P : P \in Ps : P) = \{ \text{proviso} \} \]
\[ (\exists P : P \in Ps : wp.S.P) = \{ \text{again, wp of locals} \} \]
\[ (\exists P : P \in Ps : wp." [var L ; S] ").P) \]

**RE2:** \( \text{glob}(wp." [var L ; S] ").P) \subseteq (\text{glob}(P \backslash \text{ddef}." [var L ; S] ") \cup \text{input}." [var L ; S] ") \)

provided \( \text{glob} \circ L \) and \( \text{glob}(wp.S.P) \subseteq (\text{glob}(P \backslash \text{ddef}.S) \cup \text{input}.S) \)

\[ \text{glob}(wp." [var L ; S] ").P) = \{ \text{wp of locals: } L \circ \text{glob}.P \} \]
\[ \text{glob} \circ \text{wp}.P \]
\[ \subseteq \{ \text{proviso and property of } \backslash \} \]
\[ (\text{glob}(P \backslash \text{ddef}.S) \cup \text{input}.S) \]
\[ = \{ \text{ddef of locals and set theory: } L \circ \text{glob}.P; \text{input of locals} \} \]
\[ (\text{glob}(P \backslash \text{ddef}." [var L ; S] ") \cup \text{input}." [var L ; S] ") \]

**RE3:** \( \text{wp}." [var L ; S] ").P = P \wedge \text{wp}." [var L ; S] ".true \)

provided \( \text{def}." [var L ; S] ") \circ \text{glob}.P \)

\[ \text{wp}." [var L ; S] ").P = \{ \text{wp of locals: } L \circ \text{glob}.P \} \]
\[ \text{wp}.S.P \]
\[ = \{ \text{proviso: } \text{def}.S \backslash L \circ \text{glob}.P \text{ but } L \circ \text{glob}.P \text{ so } \text{def}.S \circ \text{glob}.P \} \]
\[ P \wedge \text{wp}.true \)
\[ = \{ \text{wp of locals: } \text{glob}.true = \emptyset \} \]
\[ P \wedge \text{wp}." [var L ; S] ".true \]
APPENDIX A. FORMAL LANGUAGE DEFINITION

RE4: \( \text{ddef."}[\text{var } L ; S]\] \subseteq \text{ddef."}[\text{var } L ; S]\) provided \( \text{ddef}. S \subseteq \text{def}. S \): Indeed \( \text{ddef}. S \setminus L \subseteq \text{def}. S \setminus L \) due to the proviso and set theory.

RE5: \( \text{glob."}[\text{var } L ; S]\] = \( \text{glob."}\) \( \text{def."}[\text{var } L ; S]\] \( \cup \text{input."}[\text{var } L ; S]\) 

\[
\begin{align*}
glob."[\text{var } L ; S]\] & = \{\text{glob of locals}\} \\
(\text{def}. S \setminus L) \cup \text{input}. S & = \{\text{def and input of locals}\} \\
\text{def."}[\text{var } L ; S]\] \( \cup \text{input."}[\text{var } L ; S]\) & = \square
\end{align*}
\]

Live variables

Enclosing a statement with liveness information (e.g. \( S[\text{live } V]\)) guarantees only definitions of the live variables \( V \) may be observable on exit from \( S \). We define \( S[\text{live } V] \triangleq [\text{var } L ; S]\) where \( L := \text{def}. S \setminus V \).

Since this definition is by transformation (to another, existing language construct), there is no need to prove any of the requirements, as they can be inferred. Similarly, there is no need to define variable properties, as those can be calculated. Thus, the semantics and properties can be derived from those of local variables, as is summarised in the following. For a given statement \( S \), set of variables \( V \), a corresponding set \( L := \text{def}. S \setminus V \) and fresh \( L' \), we have:

\[
\begin{align*}
[\text{wp."} S[\text{live } V]\] . P & \equiv (\text{wp}. S.P[L \setminus L'][L' \setminus L]) \text{ for all } P \text{ with } \text{glob}. P \circ L'; \text{ or the simpler} \\
[\text{wp."} S[\text{live } V]\] . Q & \equiv \text{wp}. S.Q \text{ for all } Q \text{ with } \text{glob}. Q \circ (L, L') \\
\text{def."} S[\text{live } V] \] & \triangleq \text{def}. S \cap V \\
\text{ddef."} S[\text{live } V] \] & \triangleq \text{ddef}. S \cap V \\
\text{input."} S[\text{live } V] \] & \triangleq \text{input}. S \\
\text{glob."} S[\text{live } V] \] & \triangleq (\text{def}. S \cap V) \cup \text{input}. S 
\end{align*}
\]
Appendix B

Laws of Program Manipulation

B.1 Manipulating core statements

Law 1. Let $X, Y, E_1, E_2$ be two sets of variables and two sets of expressions, respectively; then

$$“ X := E_1 ; Y := E_2 ” = “ X, Y := E_1, E_2 ”$$

provided $X \diamond (Y \cup \text{glob}.E_2)$.

Proof. We first observe for all $P$ with $\text{glob}.P \circ \text{def}.S$

$$wp.“ X := E_1 ; Y := E_2 ” . P$$

$$= \{ \text{wp of ‘ ; ’ and twice ‘:=’} \}$$

$$(X := E_1).(Y := E_2).P$$

$$= \{ \text{merge subs: proviso} \}$$

$$(X, Y := E_1, E_2).P$$

$$= \{ \text{wp of ‘:=’} \}$$

$$wp.“ X, Y := E_1, E_2 ” . P .$$


Law 2. Let $S, X$ be a statement set of variables, respectively; then

$$S = “ S ; X := X ” .$$

Proof. We first observe for all $P$ with $\text{glob}.P \circ \text{def}.S$

$$wp.“ S ; X := X ” . P$$

$$= \{ \text{wp of ‘ ; ’ and ‘:=’} \}$$
APPENDIX B. LAWS OF PROGRAM MANIPULATION

Let $S, S_1, S_2, B$ be three statements and a boolean expression, respectively; then

\[ \text{" if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \] = \[ \text{" if } B \text{ then } S \text{ else } S \text{ fi } \]

provided $\text{def.} S \circ \text{glob.} B$.

**Proof.**

\[ \text{wp." } S ; \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \text{" } . P \]

\[ = \{ \text{wp of } \text{" ; } \text{" } \} \]

\[ \text{wp.} (\text{wp." if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \text{" } . P) \]

\[ = \{ \text{wp of IF} \} \]

\[ \text{wp.} ((B \Rightarrow \text{wp.} S_1. P) \wedge (\neg B \Rightarrow \text{wp.} S_2. P)) \]

\[ = \{ \text{wp.} S \text{ is fin. conj.} \} \]

\[ \text{wp.} (B \Rightarrow \text{wp.} S_1. P) \wedge \text{wp.} (\neg B \Rightarrow \text{wp.} S_2. P) \]

\[ = \{ \text{Lemma B.1, twice: proviso (see below)} \} \]

\[ (B \Rightarrow \text{wp.} (\text{wp.} S_1. P)) \wedge (\neg B \Rightarrow \text{wp.} S_1. \text{true}) \wedge \\ (\neg B \Rightarrow \text{wp.} (\text{wp.} S_2. P)) \wedge (B \Rightarrow \text{wp.} S_2. \text{true}) \]

\[ = \{ \text{pred. calc.} \} \]

\[ (B \Rightarrow \text{wp.} (\text{wp.} S_1. P) \wedge \text{wp.} S_1. \text{true}) \wedge (\neg B \Rightarrow \text{wp.} S_2. \text{true} \wedge \text{wp.} (\text{wp.} S_2. P)) \]

\[ = \{ \text{absorb termination 3.14 and wp of } \text{" ; } \text{" twice} \} \]
$$(B \Rightarrow \text{wp."} S ; S_1 \text{"}.P) \land (\neg B \Rightarrow \text{wp."} S ; S_2 \text{"}.P)$$

$$= \{\text{wp of IF}\}$$

$$\text{wp."} \text{if } B \text{ then } S ; S_1 \text{ else } S ; S_2 \text{ fi "}.P \ .$$

\[\square\]

**Lemma B.1.** Let $S, P, Q$ be a statement and two predicates, respectively, with $\text{def.} S \odot \text{glob.} P$; then

$$[\text{wp.} S.(P \Rightarrow Q) \equiv (P \Rightarrow \text{wp.} S.Q) \land (\neg P \Rightarrow \text{wp.} S.\text{true})] \ .$$

**Proof.**

$$\text{wp.} S.(P \Rightarrow Q)$$

$$= \{\text{pred. calc.; wp.} S \text{ is fin. disj.}\}$$

$$\text{wp.} S.(\neg P) \lor \text{wp.} S.Q$$

$$= \{\text{RE3: proviso}\}$$

$$(\neg P \land \text{wp.} S.\text{true}) \lor \text{wp.} S.Q$$

$$= \{\text{pred. calc.}\}$$

$$(\neg P \lor \text{wp.} S.Q) \land (\text{wp.} S.\text{true} \lor \text{wp.} S.Q)$$

$$= \{\text{pred. calc.; termination absorbs } 3.15\}$$

$$(P \Rightarrow \text{wp.} S.Q) \land \text{wp.} S.\text{true}$$

$$= \{\text{pred. calc.}\}$$

$$(P \Rightarrow \text{wp.} S.Q) \land (P \Rightarrow \text{wp.} S.\text{true}) \land (\neg P \Rightarrow \text{wp.} S.\text{true})$$

$$= \{\text{pred. calc.}\}$$

$$(P \Rightarrow \text{wp.} S.Q \land \text{wp.} S.\text{true}) \land (\neg P \Rightarrow \text{wp.} S.\text{true})$$

$$= \{\text{absorb termination } 3.14\}$$

$$(P \Rightarrow \text{wp.} S.Q) \land (\neg P \Rightarrow \text{wp.} S.\text{true}) \ .$$

$$\square$$

**Law 4.** Let $S_1, S_2, S_3, B$ be three statements and a boolean expression, respectively; then

“ if $B_1$ then $S_1$ else $S_2$ fi ; $S_3$ ” = “ if $B_1$ then $S_1$ ; $S_3$ else $S_2$ ; $S_3$ fi ” .

**Proof.**

$$\text{wp."} \text{if } B \text{ then } S_1 \text{ ; } S_3 \text{ else } S_2 \text{ ; } S_3 \text{ fi "}.P$$

$$= \{\text{wp of IF}\}$$
\[(B1 \implies \text{wp} \cdot X = S1 \land S3 \land P) \land (\neg B1 \implies \text{wp} \cdot X = S2 \land S3 \land P)\]

\[= \{ \text{wp of $\cdot$ ; $\cdot$, twice} \}
\]

\[(B1 \implies \text{wp} \cdot S1. (\text{wp} \cdot S3. P)) \land (\neg B1 \implies \text{wp} \cdot S2. (\text{wp} \cdot S3. P))\]

\[= \{ \text{wp of IF} \}
\]

\[\text{wp} \cdot \text{if } B1 \text{ then } S1 \text{ else } S2 \text{ fi } \cdot (\text{wp} \cdot S3. P)\]

\[= \{ \text{wp of $\cdot$ ; $\cdot$} \}
\]

\[\text{wp} \cdot \text{if } B1 \text{ then } S1 \text{ else } S2 \text{ fi } \cdot S3 \land P \]

\[\square\]

**Law 5.** Let \(S1, X, B, E\) be any statement, set of variables, boolean expression and set of expressions, respectively; then

\[\{X = E\}; \text{while } B \text { do } S1 \land (X := E) \text{ od } = \{X = E\}; \text{while } B \text { do } S1 \text{ od } \land (X := E)\]

provided \(X \circ (\text{glob} \cdot B \cup \text{input} \cdot S1 \cup \text{glob} \cdot E)\).

**Proof.**

\[\text{wp} \cdot \{X = E\}; \text{while } B \text { do } S1 \land (X := E) \land P\]

\[= \{ \text{wp of $\cdot$ ; $\cdot$, twice} \}\]

\[\text{wp} \cdot \{X = E\} \land \text{wp} \cdot \{ \text{wp} \cdot \text{while } B \text { do } S1 \land (X := E) \land P\}\]

\[= \{ \text{wp of assertions and $\cdot$ := $\cdot$} \}\]

\[(X = E) \land \text{wp} \cdot \text{while } B \text { do } S1 \land (X := E) \land P\]

\[= \{ \text{wp of DO with } k \cdot Q \equiv (B \lor (X := E) \land P) \land (\neg B \lor \text{wp} \cdot S1 \cdot Q)\}\]

\[(X = E) \land (\exists i : 0 \leq i : k \cdot \text{false})\]

\[= \{ \land \text{distributes over } \exists 3.\text{III} \}\]

\[(\exists i : 0 \leq i : (X = E) \land k \cdot \text{false})\]

\[= \{ \text{see below; } [l. Q \equiv (B \lor P) \land (\neg B \lor \text{wp} \cdot S \land (X := E) \land P)]\}\]

\[(\exists i : 0 \leq i : (X = E) \land l \cdot \text{false})\]

\[= \{ \land \text{distributes over } \exists 3.\text{III} \}\]

\[(X = E) \land (\exists i : 0 \leq i : l \cdot \text{false})\]

\[= \{ \text{wp of DO with } l \text{ as above} \}\]

\[(X = E) \land \text{wp} \cdot \text{while } B \text { do } S1 \land (X := E) \land P\]

\[= \{ \text{wp of assertions and $\cdot$ ; $\cdot$} \}\]

\[\text{wp} \cdot \{X = E\}; \text{while } B \text { do } S1 \land (X := E) \land P\]

.
We finish by proving for the middle step above, by induction, having \([X = E] \land k^i.\text{false} \equiv (X = E) \land l^i.\text{false}]\) for all \(i\), provided \(X \circ (\text{glob.B} \cup \text{input.S1} \cup \text{glob.E})\).

For the base case \(i = 0\), we observe that indeed \([X = E] \land \text{false} \equiv (X = E) \land \text{false}]\) (recall the definition of function iteration). Then, for the induction step, we assume \([X = E] \land k^i.\text{false} \equiv (X = E) \land l^i.\text{false}]\) and prove \([X = E] \land k^{i+1}.\text{false} \equiv (X = E) \land l^{i+1}.\text{false}]\).

\[
\begin{align*}
(X = E) \land k^{i+1}.\text{false} & \quad \text{(def. of func. it.)} \\
= & \quad (X = E) \land k.(k^i.\text{false}) \\
= & \quad (X = E) \land (B \lor (X := E).P) \land (\neg B \lor \text{wp.S1}.(k^i.\text{false})) \\
= & \quad \text{(replace equals with equals)} \\
= & \quad (X = E) \land (B \lor X := E).P \land (\neg B \lor \text{wp.S1}.(k^i.\text{false})) \\
= & \quad \text{(remove redundant self-sub.)} \\
= & \quad (X = E) \land (B \lor P) \land (\neg B \lor \text{wp.S1}.((X := E).(k^i.\text{false}))) \\
= & \quad \text{(intro. redundant sub.: } X \circ (k^i.\text{false}) \text{ due to RE2 and } X \circ (\text{glob.B} \cup \text{input.S1} \cup \text{glob.E}) \text{ (proviso)}) \\
= & \quad \text{(ind. hypo.)} \\
= & \quad (X = E) \land (B \lor P) \land (\neg B \lor \text{wp.S1}.((X := E).(l^i.\text{false}))) \\
= & \quad \text{(wp of } ; \text{ and } \text{:=}) \\
= & \quad (X = E) \land (B \lor P) \land (\neg B \lor \text{wp.} \text{" S1 ; } (X := E) \text{ "}.(l^i.\text{false})) \\
= & \quad \text{(def. of } l) \\
= & \quad (X = E) \land l.(l^i.\text{false}) \\
= & \quad \text{(def. of func. it.)} \\
= & \quad (X = E) \land l^{i+1}.\text{false} \
\end{align*}
\]

\(\square\)

**Law 6.** Let \(X, E\) be any set of variables and set of expressions, respectively; then

\[\text{" } \{X = E\} \text{ " } = \text{" } \{X = E\} ; X := E \text{ "} \ .\]

**Proof.**

\[\text{wp.} \text{" } \{X = E\} ; X := E \text{ "} .P\]
\[
\begin{align*}
= & \quad \{ \text{wp of } ; \text{ assertions and assignments} \} \\
& \quad (X = E) \land ((X := E).P) \\
= & \quad \{ \text{remove redundant sub.} \} \\
& \quad (X = E) \land P \\
= & \quad \{ \text{wp of assertions} \} \\
& \quad \text{wp. } \{ X = E \} \text{. } P \\
\end{align*}
\]

\[\square\]

### B.2 Assertion-based program analysis

#### B.2.1 Introduction of assertions

**Law 7.** Let \( X, Y, E_1, E_2 \) be two sets of variables and two sets of expressions, respectively; then

\[
\text{“ } X, Y := E_1, E_2 \text{ ” } = \text{ “ } X, Y := E_1, E_2 ; \{ Y = E_2 \} \text{ ”}
\]

provided \((X, Y) \diamond \text{glob.E2.}\)

**Proof.**

\[
\begin{align*}
\text{wp.} & \quad \text{“ } X, Y := E_1, E_2 ; \{ Y = E_2 \} \text{ ”. } P \\
= & \quad \{ \text{wp of } ; \text{ } \} \\
\text{wp.} & \quad \text{“ } X, Y := E_1, E_2 \text{ ”. } (\text{wp.} \{ Y = E_2 \} \text{ ”. } P \\
= & \quad \{ \text{wp of assertions} \} \\
\text{wp.} & \quad \text{“ } X, Y := E_1, E_2 \text{ ”. } ((Y = E_2) \land P) \\
= & \quad \{ \text{wp.} \text{“ } X, Y := E_1, E_2 \text{ ” is fin. conj.} \} \\
\text{wp.} & \quad \text{“ } X, Y := E_1, E_2 \text{ ”. } (Y = E_2) \land \text{wp.} \text{“ } X, Y := E_1, E_2 \text{ ”. } P \\
= & \quad \{ \text{wp of } ;\text{=}! \} \\
(X, Y := E_1, E_2).&(Y = E_2) \land \text{wp.} \text{“ } X, Y := E_1, E_2 \text{ ”. } P \\
= & \quad \{ \text{normal sub.: proviso} \} \\
& \quad (E_2 = E_2) \land \text{wp.} \text{“ } X, Y := E_1, E_2 \text{ ”. } P \\
= & \quad \{ \text{id. elem. of } \land \} \\
& \quad \text{wp.} \text{“ } X, Y := E_1, E_2 \text{ ”. } P \\
\end{align*}
\]

\[\square\]
**Law 8.** Let $X, X', E$ be (same length) lists of variables and expressions, respectively, with $X \diamond X'$; then

\[
" X, X' := E, E " = " X, X' := E, E ; \{ X = X' \} " .
\]

Proof.

\[
\begin{align*}
\wp." X, X' := E, E ; \{ X = X' \} " . & P \\
= & \{ \wp \text{ of } ; \} \\
= & \wp." X, X' := E, E \.( \wp." \{ X = X' \} " . P \\
= & \{ \wp \text{ of assertions} \} \\
= & \wp." X, X' := E, E \.( (X = X') \wedge P ) \\
= & \{ \wp \text{ is fin. conj.} \} \\
= & \wp." X, X' := E, E \.( (X = X') \wedge \wp." X, X' := E, E " . P \\
= & \{ \wp \text{ of } :=' \} \\
= & \wp." (X, X' := E, E) . (X = X') \wedge \wp." X, X' := E, E " . P \\
= & \{ \text{normal sub.: proviso} \} \\
= & \wp." X, X' := E, E " . P \\
= & \{ \text{id. elem. of } \wedge \} \\
\end{align*}
\]

**Law 9.** Let $S_1, B_1, B_2$ be any given statement and two boolean expressions, respectively; then

\[
" \text{while } B_1 \text{ do } S_1 \text{ od } " = " \text{while } B_1 \text{ do } \{ B_2 \} ; S_1 \text{ od } " .
\]

provided $[B_1 \Rightarrow B_2]$.

Proof. In order to prove that the two loop statements are equivalent, it suffices to show for all $Q$

\[
[\neg B_1 \lor \wp.S_1.Q \equiv \neg B_1 \lor \wp." \{ B_2 \} ; S_1 " .Q].
\]

\[
\begin{align*}
\neg B_1 \lor \wp." \{ B_2 \} ; S_1 " . & Q \\
= & \{ \wp \text{ of } ; \} \text{ and assertions} \\
= & \neg B_1 \lor (B_2 \land \wp.S_1.Q) \\
= & \{ \text{pred. calc.} \}
\end{align*}
\]
(¬B1 ∨ B2) ∧ (¬B1 ∨ wp.S1.Q)

= {proviso}

true ∧ (¬B1 ∨ wp.S1.Q)

= {id. elem.}

¬B1 ∨ wp.S1.Q .

\[
\begin{align*}
\text{Law 10.} \quad &\text{Let } S, B \text{ be a statement and boolean expression, respectively; then} \\
&\text{“ } \{wp.S.B\} ; S \text{ ”} = \text{“ } S ; \{B\} \text{ ”} .
\end{align*}
\]

\[\text{Proof.} \quad \text{We observe for all } P:
\]

\[
\begin{align*}
wp.\text{“ } S ; \{B\} \text{ ”}.P &= \{wp \text{ of ‘ ; ’}\} \\
wp.S.(wp.\text{“ } \{B\} \text{ ”}.P) &= \{wp \text{ of assertions}\} \\
wp.S.(B \land P) &= \{\text{conj. of } wp.S\} \\
wp.S.B \land wp.S.P) &= \{wp \text{ of assertions}\} \\
wp.\text{“ } \{wp.S.B\} \text{ ”}.(wp.S.P) &= \{wp \text{ of ‘ ; ’}\} \\
wp.\text{“ } \{wp.S.B\} ; S \text{ ”}.P &. 
\end{align*}
\]

\[
\begin{align*}
\text{Law 11.} \quad &\text{Let } S, B \text{ be a statement and boolean expression, respectively; then} \\
&\text{“ } \{B\} ; S \text{ ”} = \text{“ } S ; \{B\} \text{ ”} .
\end{align*}
\]

\[\text{provided } \text{def.} S \circ \text{glob.} B.\]

\[\text{Proof.} \quad \text{“ } S ; \{B\} \text{ ”}\]
APPENDIX B. LAWS OF PROGRAM MANIPULATION

= \{\text{swap statements (Law 5.7): def of assertions is empty and def}S \circ \text{glob.}B \text{ (proviso)}\}

\text{“ \{B\} ; S ”}.

The following law will be used for propagating assertions forward into branches of an IF as well as backward ahead of an IF.

**Law 12.** Let $S_1, S_2, B_1, B_2$ be two statements and two boolean expressions, respectively; then

\text{“ \{B_1\} ; if } B_2 \text{ then } S_1 \text{ else } S_2 \text{ fi ”} = \text{“ if } B_2 \text{ then } \{B_1\} ; S_1 \text{ else } \{B_1\} ; S_2 \text{ fi ”}.

**Proof.**

\text{“ \{B_1\} ; if } B_2 \text{ then } S_1 \text{ else } S_2 \text{ fi ”}

= \{\text{Law 3: proviso def.} \{B_1\} = \emptyset \text{ for any assertion}\}

\text{“ if } B_2 \text{ then } \{B_1\} ; S_1 \text{ else } \{B_1\} ; S_2 \text{ fi ”}.

The next law will allow the propagation of assertions forward to the (head of the) body of a loop.

**Law 13.** Let $S, B_1, B_2, B_3, B_4$ be a statement and four boolean expressions, respectively; then

\text{“ \{B_1\} ; while } B_2 \text{ do } S ; \{B_3\} \text{ od ”} = \text{“ \{B_1\} ; while } B_2 \text{ do } \{B_4\} ; S ; \{B_3\} \text{ od ”}

provided $[B_1 \Rightarrow B_4]$ and $[B_3 \Rightarrow B_4]$.

**Proof.**

\text{“ \{B_1\} ; while } B_2 \text{ do } S ; \{B_3\} \text{ od ”}

= \{\text{Law 14: proviso def} \}

\text{“ \{B_1\} ; while } B_2 \land B_4 \text{ do } S ; \{B_3\} \text{ od ”}

= \{\text{Law 9: } [B_2 \land B_4 \Rightarrow B_4]\}

\text{“ \{B_1\} ; while } B_2 \land B_4 \text{ do } \{B_4\} ; S ; \{B_3\} \text{ od ”}.

\hfill \Box
Law 14. Let \(S, B_1, B_2, B_3, B_4\) be a statement and four boolean expressions, respectively; then

\[
\text{" \{B_1\} ; \text{while } B_2 \text{ do } S ; \{B_3\} \text{ od }" = \text{" \{B_1\} ; \text{while } B_2 \land B_4 \text{ do } S ; \{B_3\} \text{ od }"}
\]

provided \([B_1 \Rightarrow B_4]\) and \([B_3 \Rightarrow B_4]\).

Proof.

\[
\text{" \{B_1\} ; \text{while } B_2 \text{ do } S ; \{B_3\} \text{ od }“}
\]

= \{proviso and pred. calc.\}

\[
\text{" \{B_1 \land B_4\} ; \text{while } B_2 \text{ do } S ; \{B_3 \land B_4\} \text{ od }“}
\]

= \{Law 15\}

\[
\text{" \{B_1\} ; \{B_4\} ; \text{while } B_2 \text{ do } S ; \{B_3\} ; \{B_4\} \text{ od }“}
\]

= \{see below\}

\[
\text{" \{B_1 \land B_4\} ; \text{while } B_2 \land B_4 \text{ do } S ; \{B_3 \land B_4\} \text{ od }“}
\]

= \{Law 15\}

\[
\text{" \{B_1\} ; \text{while } B_2 \land B_4 \text{ do } S ; \{B_3 \land B_4\} \text{ od }“}
\]

= \{proviso and pred. calc.\}

\[
\text{" \{B_1\} ; \text{while } B_2 \land B_4 \text{ do } S ; \{B_3\} \text{ od }“}
\]

So, we are left to prove the middle step above, simplified to

\[
\text{" \{B_4\} ; \text{while } B_2 \text{ do } S_1 ; \{B_4\} \text{ od }“ = \text{" \{B_4\} ; \text{while } B_2 \land B_4 \text{ do } S_1 ; \{B_4\} \text{ od }“}
\]

wp." \{B_4\} ; \text{while } B_2 \text{ do } S_1 ; \{B_4\} \text{ od }“.P

= \{wp of ‘‘ ; ‘’ and assertions\}

\(B_4 \land \text{wp." while } B_2 \text{ do } S_1 ; \{B_4\} \text{ od }“.P\)

= \{wp of DO with \([k.Q \equiv (B_2 \lor P) \land (\neg B_2 \lor \text{wp." S_1 ; \{B_4\} “} .Q)])\}

\(B_4 \land (\exists i : 0 \leq i : k^i.\text{false})\)

= \{\land distributes over \exists [3.11]\}\}

\(B_4 \land (\exists i : 0 \leq i : B_4 \land k^i.\text{false})\)

= \{see below; \([l.Q \equiv ((B_2 \land B_4) \lor P) \land (\neg (B_2 \land B_4) \lor \text{wp." S_1 ; \{B_4\} “} .Q)])\}

\(B_4 \land (\exists i : 0 \leq i : B_4 \land l^i.\text{false})\)

= \{\land distributes over \exists [3.11]\}

\(B_4 \land (\exists i : 0 \leq i : l^i.\text{false})\)
\[\begin{align*}
\text{Law 15.} \quad & \text{Let } B_1, B_2 \text{ be two boolean expressions; then} \\
& " \{B_1 \land B_2\} " = " \{B_1\} ; \{B_2\} " .
\end{align*}\]

\begin{proof}
\[\begin{align*}
\text{wp." } & \{B_1\} ; \{B_2\} " .P \\
= & \{ \text{wp of } \cdot \land \text{ and assertions}\} \\
B_1 & \land \text{wp." } \{B_2\} " .P \\
= & \{ \text{wp of assertions}\} \\
B_1 \land B_2 & \land P \\
= & \{ \text{wp of assertions}\} \\
\text{wp." } & \{B_1 \land B_2\} " .P .
\end{align*}\]
\end{proof}
Law 16. Let $S, B_1, B_2$ be a statement and two boolean expressions, respectively; then

$$\{B_1\} ; \text{while } B_2 \text{ do } S \text{ od } = \{B_1\} ; \text{while } B_2 \{B_1\} ; S \text{ od }$$

provided $\text{glob.}B_1 \circ \text{def.}S$.

Proof.

\[
\text{wp."} \{B_1\} ; \text{while } B_2 \{B_1\} ; S \text{ od } \cdot P
= \{\text{wp of } ' ; ' \text{ and assertions}\}
B_1 \land \text{wp."} \text{while } B_2 \{B_1\} ; S \text{ od } \cdot P
= \{\text{Law 11} \} \text{glob.}B_1 \circ \text{def.}S \text{ (proviso)}
B_1 \land \text{wp."} \text{while } B_2 \{B_1\} \text{ od } \cdot P
= \{\text{wp of DO with } [k.Q \equiv (B_2 \lor P) \land (\neg B_2 \lor \text{wp."} S ; \{B_1\} \cdot Q)]\}
B_1 \land (\exists i : 0 \leq i : k^i.\text{false})
= \{\land \text{distributes over } \exists \{3.11\}\}
B_1 \land (\exists i : 0 \leq i : B_1 \land k^i.\text{false})
= \{\text{see below; } [l.Q \equiv (B_2 \lor P) \land (\neg B_2 \lor \text{wp.S} . Q)]\}
B_1 \land (\exists i : 0 \leq i : B_1 \land l^i.\text{false})
= \{\land \text{distributes over } \exists \{3.11\}\}
B_1 \land (\exists i : 0 \leq i : l^i.\text{false})
= \{\text{wp of DO with } l \text{ as above}\}
B_1 \land \text{wp."} \text{while } B_2 \{B_1\} ; S \text{ od } \cdot P
= \{\text{wp of assertions and } ' ; ' \}
\text{wp."} \{B_1\} ; \text{while } B_2 \{B_1\} ; S \text{ od } \cdot P .
\]

We finish by proving for the middle step above, by induction, having $[B_1 \land k^i.\text{false} \equiv B_1 \land l^i.\text{false}]$ for all $i$, provided $\text{glob.}B_1 \circ \text{def.}S$.

For the base case $i = 0$, we observe that indeed $[B_1 \land \text{false} \equiv B_1 \land \text{false}]$ (recall the definition of function iteration). Then, for the induction step, we assume $[B_1 \land k^i.\text{false} \equiv B_1 \land l^i.\text{false}]$ and prove $[B_1 \land k^{i+1}.\text{false} \equiv B_1 \land l^{i+1}.\text{false}]$.

\[
B_1 \land k^{i+1}.\text{false}
= \{\text{def. of func. it.}\}
B_1 \land k.(k^i.\text{false})
\]
\[ B_1 \land (B_2 \lor P) \land (\neg B_2 \lor \text{wp.} \{S\} \cdot (k^i \cdot \text{false})) \]

\[ B_1 \land (B_2 \lor P) \land (\neg B_2 \lor \text{wp.} (B_1 \land k^i \cdot \text{false})) \]

\[ B_1 \land (B_2 \lor P) \land (\neg B_2 \lor \text{wp.} (B_1 \land l^i \cdot \text{false})) \]

\[ \text{RE3: proviso} \]

\[ B_1 \land (B_2 \lor P) \land (\neg B_2 \lor (B_1 \land \text{wp.} S \cdot \text{true} \land \text{wp.} (l^i \cdot \text{false}))) \]

\[ \text{pred. calc.: absorption} \]

\[ B_1 \land l.(l^i \cdot \text{false}) \]

\[ \text{def. of func. it.} \]

\[ B_1 \land l^{i+1}.\text{false} \]

\[ \square \]

**B.2.3 Substitution**

**Law 17.** Let \( S_1, S_2, B \) be two statements and a boolean expression, respectively; let \( X, E \) be a set of variables and a corresponding list of expressions; and let \( Y, Y' \) be two sets of variables; then

\[ \text{"} \{ Y = Y' \} ; X := E \text{"} = \text{"} \{ Y = Y' \} ; X := E[Y \setminus Y'] \text{"} ; \]

\[ \text{"} \{ Y = Y' \} ; \text{IF} \text{"} = \text{"} \{ Y = Y' \} ; \text{IF}' \text{"} ; \text{and} \]

\[ \text{"} \{ Y = Y' \} ; \text{DO} \text{"} = \text{"} \{ Y = Y' \} ; \text{DO}' \text{"} . \]
where IF := “ if B then S1 else S2 fi ”,
IF′ := “ if B[Y \ Y′] then S1 else S2 fi ”,
DO := “ while B do S1 ; { Y = Y′} od ”
and DO′ := “ while B[Y \ Y′] do S1 ; { Y = Y′} od ”.

Proof. Assignment:

wp." { Y = Y′} ; X := E[Y \ Y′] ".P
= { wp of ' ; ' and assertions}
  (Y = Y′) ∧ wp." X := E[Y \ Y′] ".P
= { wp of ':='}
  (Y = Y′) ∧ (X := E[Y \ Y]).P
= { replace equals for equals }
  (Y = Y′) ∧ (X := E[Y \ Y]).P
= { redundant self sub. }
  (Y = Y′) ∧ (X := E).P
= { wp of ':='}
  (Y = Y′) ∧ wp." X := E ".P
= { wp of assertions and ' ; '}
wp." { Y = Y′} ; X := E ".P

IF:

wp." { Y = Y′} ; if B[Y \ Y′] then S1 else S2 fi ".P
= { wp of ' ; ' and assertions }
  (Y = Y′) ∧ wp." if B[Y \ Y′] then S1 else S2 fi ".P
= { wp of IF }
  (Y = Y′) ∧ (B[Y \ Y′] ⇒ wp.S1.P) ∧ (¬B[Y \ Y′] ⇒ wp.S2.P)
= { replace equals for equals }
= { redundant self sub., twice }
  (Y = Y′) ∧ (B ⇒ wp.S1.P) ∧ (¬B ⇒ wp.S2.P)
= { wp of IF }
  (Y = Y′) ∧ wp." if B then S1 else S2 fi ".P
= { wp of assertions and ' ; '}
wp."{ Y = Y' } ; if B then S1 else S2 fi " . P .

DO:

wp."{ Y = Y' } ; while B do S1 ; { Y = Y' } od " . P

=  { wp of \( \cdot \) ; \( \cdot \) and assertions }

( Y = Y' ) \& wp." while B do S1 ; { Y = Y' } od " . P

=  { wp of DO with \([k.Q \equiv (B \lor P) \& (\neg B \lor wp." S1 ; { Y = Y' } \) \). Q\] with

( Y = Y' ) \& ( \exists i : 0 \leq i : k^i . false )

=  { \& distributes over \( \exists \ (3.11) \) }

( \exists i : 0 \leq i : ( Y = Y' ) \& l^i . false )

=  { \& distributes over \( \exists \ (3.11) \) }

( Y = Y' ) \& ( \exists i : 0 \leq i : l^i . false )

=  { wp of DO with \( l \) as above }

( Y = Y' ) \& wp." while B[Y \ Y'] do S1 ; { Y = Y' } od " . P

=  { wp of assertions and \( \cdot \) ; \( \cdot \) }

wp."{ Y = Y' } ; while B[Y \ Y'] do S1 ; { Y = Y' } od " . P .

We finish by proving for the middle step above, by induction, having \( [( Y = Y' ) \& ( k^i . false \equiv ( Y = Y' ) \& l^i . false )] \) for all \( i \).

For the base case \( i = 0 \), we observe that indeed \( [( Y = Y' ) \& false \equiv ( Y = Y' ) \& false ] \) (recall the definition of function iteration). Then, for the induction step, we assume \( [( Y = Y' ) \& k^i . false \equiv ( Y = Y' ) \& l^i . false ] \) and prove \( [( Y = Y' ) \& k^{i+1} . false \equiv ( Y = Y' ) \& l^{i+1} . false ] \).
\begin{align*}
\text{Law 18.} \quad \text{Let } S_1, S_2, B \text{ be two statements and a boolean expression, respectively; let } X, X', Y, Z, E_1, E_1', E_2, E_3 \text{ be four lists of variables and corresponding lists of expressions; then } \\
\text{“} X, Y := E_1, E_2 ; Z := E_3 \text{” } = \text{ “} X, Y := E_1, E_2 ; Z := E_3[Y \setminus E_2] \text{” } ; \\
\text{“} X, Y := E_1, E_2 ; IF \text{” } = \text{ “} X, Y := E_1, E_2 ; IF' \text{” } ; \text{ and } \\
\text{“} X, Y := E_1, E_2 ; DO \text{” } = \text{ “} X, Y := E_1, E_2 ; DO' \text{” } \\
\text{provided } ((X \cup X'), Y) \circ \text{glob.E2} \\
\text{where } \text{IF} := \text{ “ if B then S_1 else S_2 fi ” } , \\
\text{IF'} := \text{ “ if B[Y \setminus E_2] then S_1 else S_2 fi ” } , \\
\text{DO} := \text{ “ while B do S_1 ; X', Y := E_1', E_2 od ” } \\
\text{and } \text{DO'} := \text{ “ while B[Y \setminus E_2] do S_1 ; X', Y := E_1', E_2 od ” } . \\
\text{Proof.} \\
\text{“} X, Y := E_1, E_2 ; Z := E_3 \text{” } \\
= \text{ {intro. following assertion (Law 7): (X, Y) \circ \text{glob.E2} (proviso)} } \\
\text{“} X, Y := E_1, E_2 ; \{ Y = E_2 \} ; Z := E_3 \text{” } \\
= \text{ {assertion-based sub. (Law 17) with } Y' := E_2 } \\
\text{“} X, Y := E_1, E_2 ; \{ Y = E_2 \} ; Z := E_3[Y \setminus E_2] \text{” } \\
= \text{ {remove following assignment (Law 7)} } \\
\text{“} X, Y := E_1, E_2 ; Z := E_3[Y \setminus E_2] \text{” } .
\end{align*}
“ $X, Y := E_1, E_2$ ; if $B$ then $S_1$ else $S_2$ fi ”

=  {intro. following assertion (Law 7): $(X, Y) \diamond \text{glob.} E_2$ (proviso)}

“ $X, Y := E_1, E_2$ ; $\{ Y = E_2 \}$ ; if $B$ then $S_1$ else $S_2$ fi ”

=  {assertion-based sub. (Law 17) with $Y' := E_2$}

“ $X, Y := E_1, E_2$ ; if $B[Y \setminus E_2]$ then $S_1$ else $S_2$ fi ”

=  {remove following assignment (Law 7)}

“ $X, Y := E_1, E_2$ ; while $B$ do $S_1$ ; $X', Y := E_1', E_2$ od ”

=  {intro. following assertion (Law 7), twice:
   $(X, Y) \diamond \text{glob.} E_2$ and $(X', Y) \diamond \text{glob.} E_2$ (proviso)}

“ $X, Y := E_1, E_2$ ; $\{ Y = E_2 \}$ ; while $B$ do $S_1$ ; $X', Y := E_1', E_2$ ; $\{ Y = E_2 \}$ od ”

=  {assertion-based sub. (Law 17) with $Y' := E_2$}

“ $X, Y := E_1, E_2$ ; while $B[Y \setminus E_2]$ do $S_1$ ; $X', Y := E_1', E_2$ od ”

=  {remove following assignment (Law 7), twice}

“ $X, Y := E_1, E_2$ ; while $B[Y \setminus E_2]$ do $S_1$ ; $X', Y := E_1', E_2$ od ”

\[\square\]

B.3 Live variables analysis

B.3.1 Introduction and removal of liveness information

Law 19. Let $S, V$ be any statement and set of variables, respectively, with $\text{def.} S \subseteq V$; then

$S = " S[\text{live } V] "$ .

Proof. We observe for all $Q$

wp." $S[\text{live } V]$ ".$Q$

=  {def. of $\text{live}$ with $\text{coV} := \text{def.} S \setminus V$}

wp." $[\text{var } \text{coV} ; S]"$.$Q$

=  {wp of locals: $\text{coV} = \emptyset$ due to proviso}
APPENDIX B. LAWS OF PROGRAM MANIPULATION

B.3.2 Propagation of liveness information

Law 20. Let $S_1, S_2, V_1, V_2$ be any two statements and two sets of variables, respectively; then

$$(S_1 ; S_2)[live \ V_1] = (S_1[live \ V_2] ; S_2[live \ V_1])[live \ V_1]$$

provided $V_2 = (V_1 \setminus ddef.S_2) \cup input.S_2$.

Proof. We observe for all $P$ (with $glob.P \subseteq V_1$)

$$wp.(S_1[live \ V_2] ; S_2[live \ V_1])[live \ V_1].P$$

$$= \{ wp \ of \ live : proviso \} wp.(S_1[live \ V_2] ; S_2[live \ V_1]).P$$

$$= \{ wp \ of \ ; \} wp.(S_1[live \ V_2]).(wp.(S_2).P)$$

$$= \{ wp \ of \ glob. \ P \subseteq V_1 \} wp.(S_1[live \ V_2]).(wp.S_2.P)$$

$$= \{ wp \ of \ glob.(wp.S_2.P) \subseteq (V_1 \setminus ddef.S_2) \cup input.S_2 \}$$

due to RE2 and the proviso

$$wp.S_1.(wp.S_2.P)$$

$$= \{ wp \ of \ ; \} wp.(S_1 ; S_2).P$$

$$= \{ wp \ of \ live : glob. \ P \subseteq V_1 \} wp.(S_1 ; S_2[live \ V_1]).P .$$

Law 21. Let $B, S_1, S_2, V$ be any boolean expression, two statements and set of variables, respectively; then

$$(if \ B \ then \ S_1 \ else \ S_2 \ fi)[live \ V] = (if \ B \ then \ S_1[live \ V] \ else \ S_2[live \ V] \ fi)[live \ V] .$$

Proof. We observe for all $P$ with $glob.P \subseteq V$

$$wp.(if \ B \ then \ S_1[live \ V] \ else \ S_2[live \ V] \ fi)[live \ V].P$$

$$= \{ wp \ of \ live : glob. \ P \subseteq V \}$$
wp." if \( B \) then \( S_1[\text{live } V] \) else \( S_2[\text{live } V] \) fi ".\( P \)

\[
\begin{align*}
= & \quad \{ \text{wp of IF} \} \\
& \quad (B \Rightarrow \text{wp." } S_1[\text{live } V] \text{ ".}P) \land (\neg B \Rightarrow \text{wp." } S_2[\text{live } V] \text{ ".}P) \\
= & \quad \{ \text{wp of live, twice: glob.} P \subseteq V \} \\
& \quad (B \Rightarrow \text{wp.} S_1. P) \land (\neg B \Rightarrow \text{wp.} S_2. P) \\
= & \quad \{ \text{wp of IF} \} \\
\end{align*}
\]

wp." if \( B \) then \( S_1 \) else \( S_2 \) fi ".\( P \)

\[
\begin{align*}
= & \quad \{ \text{wp of live: glob.} P \subseteq V \} \\
\end{align*}
\]

wp." (if \( B \) then \( S_1 \) else \( S_2 \)[live \( V] \) ".\( P \).

\[
\square
\]

**Law 22** Let \( B, S, V \) be any boolean expression, statement and set of variables, respectively; then

" (while \( B \) do \( S \) od)[live \( V1] \) " = " (while \( B \) do \( S[\text{live } V2] \) od)[live \( V1] \)

provided \( V_2 = V_1 \cup (\text{glob.} B \cup \text{input.} S) \).

**Proof.** We observe for all \( P \) with glob.\( P \subseteq V_1 \)

\[
\begin{align*}
\text{wp." (while } B \text{ do } S \text{ od}[\text{live } V1] \) ".\( P \\
= & \quad \{ \text{wp of live: glob.} P \subseteq V_1 \} \\
\text{wp." while } B \text{ do } S[\text{live } V2] \text{ od ]}.\( P \\
= & \quad \{ \text{wp of DO with } [k. Q \equiv (B \lor P) \land (\neg B \lor \text{wp.} S[\text{live } V2]. Q)] \} \\
(\exists i : 0 \leq i : k^i. \text{false}) \\
= & \quad \{ \text{see below: } [l. Q \equiv (B \lor P) \land (\neg B \lor \text{wp.} S. Q)] \} \\
(\exists i : 0 \leq i : l^i. \text{false}) \\
= & \quad \{ \text{wp of DO with } l \text{ as above} \} \\
\text{wp." while } B \text{ do } S \text{ od ]}.\( P \\
= & \quad \{ \text{wp of live: glob.} P \subseteq V_1 \} \\
\text{wp." (while } B \text{ do } S \text{ od}[\text{live } V1] \) ".\( P \\

\end{align*}
\]

We go on by proving for the missing step above \( [k^i. \text{false} \equiv l^i. \text{false}] \) for all \( i \). We begin by observing \( \text{wp.} S. Q \equiv \text{wp." } S[\text{live } V2] \text{ ".} Q \) for all \( Q \) with glob.\( P \subseteq V_2 \) due to wp of live . And indeed glob.\( ((B \lor P) \land (\neg B \lor \text{wp.} S. Q)) \subseteq V_2 \) for all such \( Q \). This is due to the proviso \( V_2 = V_1 \cup (\text{glob.} B \cup \text{input.} S) \), the given glob.\( P \subseteq V_1 \), and RE2.


B.3.3 Dead assignments: introduction and elimination

**Law 23.** Let $S, V, X, Y, E_1, E_2$ be any statement, three sets of variables and two sets of expressions, respectively; then

\[
\{ S ; X := E_1 \} \text{ live } V \} = \{ S ; X, Y := E_1, E_2 \} \text{ live } V \}
\]

provided $Y \circ (X \cup V)$.

**Proof.** We observe for all $P$ with $\text{glob}.P \subseteq V$

\[
wp.\{ S ; X, Y := E_1, E_2 \}.P
\]

\[
= \{ \text{wp } ; \cdot \text{ and } \cdot \cdot \cdot \}
\]

\[
wp.\text{S.P}[X, Y \setminus E_1, E_2]
\]

\[
= \{ \text{remove redundant sub. } Y \circ \text{glob}.P \}
\]

\[
wp.\text{S.P}[X \setminus E_1]
\]

\[
= \{ \text{wp } ; \cdot \text{ and } \cdot \cdot \cdot \}
\]

\[
wp.\{ S ; (X := E_1) \}.P
\]


**Law 24.** Let $S, V, Y, E$ be any statement, two sets of variables and set of expressions, respectively; then

\[
\{ S [\text{live } V] \} = \{ S ; Y := E [\text{live } V] \}
\]

provided $Y \circ V$.

**Proof.**

\[
\{ S ; Y := E [\text{live } V] \}
\]

\[
= \{ \text{Law 23 with } X := \emptyset \}
\]

\[
\{ S [\text{live } V] \}
\]


**Law 25.** Let $S, V, X, Y, E_1, E_2$ be any statement, three sets of variables and two sets of expressions, respectively; then

\[
\{ (X := E_1 ; S) [\text{live } V] \} = \{ (X, Y := E_1, E_2 ; S) [\text{live } V] \}
\]

provided $Y \circ (X \cup (V \setminus \text{ddef}.S) \cup \text{input}.S)$.
Proof. We observe for all $P$ with $\text{glob}. P \subseteq V$

$$\text{wp}^{"} X, Y := E_1, E_2 ; S " P$$

$$= \{\text{wp} ^{`} ; ^{`} \text{ and } ^{`} := ^{`}\}$$

$$(\text{wp}. S.P)[X, Y \setminus E_1, E_2]$$

$$= \{\text{remove redundant sub.: } Y \circ \text{glob}. (\text{wp}. S.P) \text{ due to proviso and } \text{RE2}\}$$

$$(\text{wp}. S.P)[X \setminus E_1]$$

$$= \{\text{wp} ^{`} := ^{`} \text{ and } ^{`} ; ^{`}\}$$

$$\text{wp}^{"} X := E_1 ; S " P .$$

\[\square\]

Law. Let $B, S_1, S_2, Y, V, E$ be a boolean expression, two statements, two sets of variables and a set of expressions, respectively; then

$$\text{" (} S_1 ; \text{while } B \text{ do } S_2 \text{ od)[live } V \text{"") } = \text{" (} S_1 ; \text{while } B \text{ do } S_2 \text{ ; } (Y := E) \text{ od)[live } V \text{"") }$$

provided $Y \circ (V \cup \text{glob}. B \cup \text{input}. S_2)$.

Proof. We observe for all $P$ with $\text{glob}. P \subseteq V$

$$\text{wp}. S_1 ; \text{while } B \text{ do } S_2 \text{ ; } (Y := E) \text{ od]["}. P$$

$$= \{\text{wp} \text{ of } ^{`} ; ^{`}\}$$

$$\text{wp}. S_1. (\text{wp}^{"} \text{ while } B \text{ do } S_2 \text{ ; } (Y := E) \text{ od } " \text{}. P)$$

$$= \{\text{wp} \text{ of DO with } [k.Q \equiv (B \lor P) \land (\neg B \lor \text{wp}^{"} S_2 \text{ ; } (Y := E) \text{ "}. Q)]\}$$

$$\text{wp}. S_1. (\exists i : 0 \leq i : k^i. \text{false})$$

$$= \{\text{see below: } [l.Q \equiv (B \lor P) \land (\neg B \lor \text{wp}. S_2.Q)]\}$$

$$(\text{wp}. S_1. (\exists i : 0 \leq i : l^i. \text{false})$$

$$= \{\text{wp} \text{ of DO with } l \text{ as above}\}$$

$$\text{wp}. S_1. (\text{wp}^{"} \text{ while } B \text{ do } S_2 \text{ od } " \text{}. P)$$

$$= \{\text{wp} ^{`} ; ^{`}\}$$

$$\text{wp}^{"} S_1 ; \text{while } B \text{ do } S_2 \text{ od } " \text{}. P .$$

We go on by proving for the middle step above $[k^i. \text{false} \equiv l^i. \text{false}]$ for all $i$. Keeping in mind $(\text{glob}. (P, B) \cup \text{input}. S_2) \circ Y$, we begin by observing $\text{glob}. (\text{wp}. S_2.Q) \circ Y$ for all $Q$ with $\text{glob}. Q \circ Y$. Thus $\text{glob}. (k^i. \text{false}) \circ Y$ for all $i$. We now complete the proof by showing $[\text{wp}^{"} S_2 ; (Y := E) " \text{. Q } \equiv \text{wp}. S_2.Q]$ for all such $Q$. 
\begin{align*}
\text{wp.} & \quad \text{"} S_2 ; (Y := E) \text{"}.Q \\
= & \quad \{\text{wp of } ; \text{ } \text{ and assignment}\} \\
\text{wp.} & \quad S_2.Q[Y \setminus E] \\
= & \quad \{\text{remove redundant sub.: proviso}\} \\
\text{wp.} & \quad S_2.Q.
\end{align*}
Appendix C

Properties of Slides

Lemma C.1. Let $S$ be any core statement and let $V_1, V_2$ be two sets of variables with $V_2 \circ (V_1 \cup \text{def}.S)$; then
\[
\text{slides}.S.V_1 = \text{slides}.S.(V_1, V_2).
\]

Proof. We prove the equivalence by induction on the structure of $S$.

First, when $V_1 \circ \text{def}.S$ we get

\[
\text{slides}.S.V_1 = \{\text{def. of slides}\}
\]

\[
skip = \{\text{def. of slides: } (V_1, V_2) \circ \text{def}.S\}
\]

\[
\text{slides}.S.(V_1, V_2).
\]

In the remaining cases we can assume $V_1 \cap \text{def}.S \neq \emptyset$.

$S = \text{"X := E";}$

\[
\text{slides." X := E ".(V_1, V_2)}
\]

\[
= \{\text{slides of ':'; } X_1 := X \cap (V_1, V_2);
\text{let } E_1 \text{ be the subset of } E \text{ corresponding to } X_1\}
\]

\[
\text{" X_1 := E_1 "}
\]

\[
= \{\text{slides of ':'; } X_1 = X \cap V_1 \text{ due to } V_2 \circ X\}
\]

\[
\text{slides." X := E ",V_1}.
\]

$S = \text{"S1 ; S2";}$

\[
\text{slides." S1 ; S2 ".(V_1, V_2)}.
\]

180
APPENDIX C. PROPERTIES OF SLIDES

\begin{align*}
\text{= } & \{\text{slides of } '; '\} \\
\text{\quad " (slides.} S_1. (V_1, V_2)) \; \text{; (slides.} S_2. (V_1, V_2)) \text{"} \\
\text{= } & \{\text{ind. hypo., twice: def.} S_1 \subseteq \text{def.} S_1 \; ; \; S_2 \text{"; similarly for } S_2\} \\
\text{\quad " (slides.} S_1. V_1) \; \text{; (slides.} S_2. V_1) \text{"} \\
\text{= } & \{\text{slides of } ' ; '\} \\
\text{\quad slides." } S_1 \; ; \; S_2 \text{" } . \ V_1 .
\end{align*}

\[ S = \text{" if } B \text{ then } S_1 \text{ else } S_2 \text{ fi ":} \]
\begin{align*}
\text{\quad \text{slides." if } B \text{ then } S_1 \text{ else } S_2 \text{ fi ", (V_1, V_2) } } \\
\text{= } & \{\text{slides of IF}\} \\
\text{\quad " if } B \text{ then slides.} S_1. (V_1, V_2) \text{ else slides.} S_2. (V_1, V_2) \text{ fi "} \\
\text{= } & \{\text{ind. hypo., twice: def.} S_1 \subseteq \text{def.} IF; \text{ similarly for } S_2\} \\
\text{\quad " if } B \text{ then slides.} S_1. V_1 \text{ else slides.} S_2. V_1 \text{ fi "} \\
\text{= } & \{\text{slides of IF}\} \\
\text{\quad slides." if } B \text{ then } S_1 \text{ else } S_2 \text{ fi ", } . \ V_1 .
\end{align*}

\[ S = \text{" while } B \text{ do } S_1 \text{ od ":} \]
\begin{align*}
\text{\quad \text{slides." while } B \text{ do } S_1 \text{ od ", (V_1, V_2) } } \\
\text{= } & \{\text{slides of DO}\} \\
\text{\quad " while } B \text{ do (slides.} S_1. (V_1, V_2)) \text{ od "} \\
\text{= } & \{\text{ind. hypo.}\} \\
\text{\quad while } B \text{ do (slides.} S_1. V_1) \text{ od } \\
\text{= } & \{\text{slides of DO: def.} S_1 = \text{def.} DO\} \\
\text{\quad slides." while } B \text{ do } S_1 \text{ od ", } . \ V_1 .
\end{align*}

\[ \square \]

**Theorem 8.1 (Slides Distribute over Union).** Any pair of slides of a single core statement, \( \text{slides.} S. V_1 \) and \( \text{slides.} S. V_2 \), is unifiable. Furthermore, we have
\[ \text{\begin{align*}
\text{(slides.} S. (V_1 \cup V_2)) &= ((\text{slides.} S. V_1) \cup (\text{slides.} S. V_2)) . 
\end{align*}} \]

*Proof.* We prove the equivalence by induction on the structure of \( S \).

First, when \( V_1 \diamond \text{def.} S \) we get
\[ \text{\begin{align*}
\text{\quad \text{slides.} } S. V_1 \cup \text{slides.} S. V_2
\end{align*}} \]
APPENDIX C. PROPERTIES OF SLIDES

\[ \{\text{def. of slides when } \text{def.} \circ V1\} \]

\[ \text{skip} \cup \text{slides}.S.V2 \]

\[ \{\text{def. of } \cup\} \]

\[ \text{slides}.S.V2 \]

\[ \{\text{Lemma C.1 } (V1 \setminus V2) \circ \text{def.} S\} \]

\[ \text{slides}.S.(V1 \cup V2) \]

A similar derivation proves the case of \( V2 \circ \text{def.} S \). Thus in the remaining cases we are left to assume both \( V1 \cap \text{def.} S \neq \emptyset \) and \( V2 \cap \text{def.} S \neq \emptyset \).

\[ S = " X := E " : \]

\[ \text{slides}." X := E " \cup \text{slides}." X := E " \cdot V2 \]

\[ \{\text{slides of } := \text{, twice: } X1 := X \cap V1, X2 := X \cap V2 \text{ and } \} \]

\[ E1, E2 \text{ are the corresponding subsets of } E \}

\[ " X1 := E1 " \cup " X2 := E2 " \]

\[ \{\text{def. of } \cup \text{ for assignments: } X12 := X1 \cup X2 \text{ and } \} \]

\[ E12, \text{ the corresponding union of } E1 \text{ and } E2 \text{ is well defined: } \}

\[ \text{any } X.i \text{ in } X12 \text{ that is both in } X1 \text{ and } X2 \text{ is also in } X \text{ and so both } \]

\[ E1.i \text{ and } E2.i \text{ are the same (original) } E.i \}\]

\[ " X12 := E12 " \]

\[ \{\text{slides of } := \text{ and set theory: } (X \cap V1) \cup (X \cap V2) = X \cap (V1 \cup V2)\} \]

\[ \text{slides}." X := E " \cdot (V1 \cup V2) \]

\[ S = " S1 ; S2 " : \]

\[ \text{slides}." S1 ; S2 " \cdot V1 \cup \text{slides}." S1 ; S2 " \cdot V2 \]

\[ \{\text{slides of } ; \text{, twice}\} \]

\[ " (\text{slides}.S1.V1) ; (\text{slides}.S2.V1) " \cup \]

\[ " (\text{slides}.S1.V2) ; (\text{slides}.S2.V2) " \]

\[ \{\text{def. of } \cup \text{ for } ; \text{ }\} \]

\[ " ((\text{slides}.S1.V1) \cup (\text{slides}.S1.V2)) ; ((\text{slides}.S2.V1) \cup (\text{slides}.S2.V2)) " \]

\[ \{\text{ind. hypo., twice}\} \]

\[ " (\text{slides}.S1.(V1 \cup V2)) ; (\text{slides}.S2.(V1 \cup V2)) " \]

\[ \{\text{slides of } ; \text{ }\} \]

\[ \text{slides}." S1 ; S2 " \cdot (V1 \cup V2) \].
APPENDIX C. PROPERTIES OF SLIDES

183

\[ S = " \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi }" : \]

\[ \text{slides}." \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi }".V_1 \cup \text{slides}." \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi }".V_2 \]

\[ = \{ \text{slides of IF, twice} \} \]

\[ " \text{if } B \text{ then slides}.S_1.V_1 \text{ else slides}.S_2.V_1 \text{ fi }" \cup " \text{if } B \text{ then slides}.S_1.V_2 \text{ else slides}.S_2.V_2 \text{ fi }" \]

\[ = \{ \text{def. of } \cup \text{ for IF} \} \]

\[ " \text{if } B \text{ then } ((\text{slides}.S_1.V_1) \cup (\text{slides}.S_1.V_2)) \text{ else } ((\text{slides}.S_2.V_1) \cup (\text{slides}.S_2.V_2)) \text{ fi }" \]

\[ = \{ \text{slides of IF} \} \]

\[ \text{slides}." \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi }".(V_1 \cup V_2) . \]

\[ S = " \text{while } B \text{ do } S_1 \text{ od }" : \]

\[ \text{slides}." \text{while } B \text{ do } S_1 \text{ od }".V_1 \cup \text{slides}." \text{while } B \text{ do } S_1 \text{ od }".V_2 \]

\[ = \{ \text{slides of DO, twice} \} \]

\[ " \text{while } B \text{ do } (\text{slides}.S_1.V_1) \text{ od }" \cup " \text{while } B \text{ do } (\text{slides}.S_1.V_2) \text{ od }" \]

\[ = \{ \text{def. of } \cup \text{ for DO} \} \]

\[ \text{while } B \text{ do } ((\text{slides}.S_1.V_1) \cup (\text{slides}.S_1.V_2)) \text{ od } \]

\[ = \{ \text{ind. hypo.} \} \]

\[ \text{while } B \text{ do } (\text{slides}.S_1.(V_1 \cup V_2)) \text{ od } \]

\[ = \{ \text{slides of DO} \} \]

\[ \text{slides}." \text{while } B \text{ do } S_1 \text{ od }".(V_1 \cup V_2) . \]

\[ \square \]

**Lemma 9.5** Let \( S, V \) be any core statement and set of variables, respectively; then

\[ \text{def}.(\text{slides}.S,V) \subseteq \text{def}.S \ . \]

**Proof.** The proof is by induction on the structure of \( S \).

First, when \( V \odot \text{def}.S \) we get

\[ \text{def}.(\text{slides}.S,V) \]

\[ = \{ \text{def. of } \text{slides: } V \odot \text{def}.S \} \]

\[ \text{def}.(\text{skip}) \]

\[ = \{ \text{def of } \text{skip} \} \]
∅ \subseteq \{ \text{set theory} \} \\
\text{def} \ S \\

In the remaining cases we can assume \( V \cap \text{def} \ S \neq \emptyset \).

\( S = " \ X := E " : \\
\text{def}(\text{slides." } X := E \ " .V) \\
= \{ \text{slides of } ':=': X1 := X \cap V; \text{let } E1 \text{ be the subset of } E \text{ corresponding to } X1 \} \\
\text{def." } X1 := E1 " \\
= \{ \text{def of assignments} \} \\
X1 \\
\subseteq \{ \text{def. of } X1 \text{ and set theory} \} \\
X \\
= \{ \text{def of assignments} \} \\
\text{def." } X := E " . \\

S = " \ S1 ; S2 " : \\
\text{def}(\text{slides." } S1 ; S2 \ " .V) \\
= \{ \text{slides of } '::' \} \\
\text{def." } (\text{slides}.S1.V) ; (\text{slides}.S2.V) " \\
= \{ \text{def of } '::' \} \\
\text{def}(\text{slides}.S1.V) \cup \text{def}(\text{slides}.S2.V) \\
\subseteq \{ \text{ind. hypo., twice} \} \\
\text{def} \ S1 \cup \text{def} \ S2 \\
= \{ \text{def of } '::' \} \\
\text{def." } S1 ; S2 " . \\

S = " \text{if } B \text{ then } S1 \text{ else } S2 \text{ fi } ": \\
\text{def}(\text{slides." if } B \text{ then } S1 \text{ else } S2 \text{ fi } " .V) \\
= \{ \text{slides of } \text{IF} \} \\
\text{def." if } B \text{ then } \text{slides}.S1.V \text{ else } \text{slides}.S2.V \text{ fi } " \\
= \{ \text{def of } \text{IF} \}
C.1 Lemmata for proving independent slides yield slices

Lemma 9.3. Let S be any core statement; let T be any slip of S and let V be any set of variables; then

\[ \text{glob.}(\text{slides}.T.V) \subseteq \text{glob.}(\text{slides}.S.V) \]

Proof. The proof is by induction over the structure of S.

First, when \( V \Join \text{def}.T \) we have \( \text{slides}.T.V = \text{skip} \) and the inclusion is trivial. Hence, in the remaining cases we shall assume \( V \cap \text{def}.T \neq \emptyset \) and the implied \( V \cap \text{def}.S \neq \emptyset \), due to \( \text{def}.T \subseteq \text{def}.S \) (Lemma 9.4).

\( S = \text{" while } B \text{ do } S1 \text{ od } \): Here, \( T \) must be \( X := E \), and the inclusion is trivial.

This will be the case whenever \( T \) is \( S \) itself. Hence, in the remaining cases we shall assume \( T \) is a proper slip of \( S \).

\( S = \text{" } S1 ; S2 \text{ "} : \)

\[ \text{glob.}(\text{slides}.\text{" } S1 ; S2 \text{ "}.V) \]

\[ = \{ \text{slides of } ; \} \]
\[
glob((\text{slides}.S1.V) \ ; \ (\text{slides}.S2.V)) \\
= \ \{\text{glob of } ' ; '\} \\
glob.\text{(slides}.S1.V) \cup \glob.\text{(slides}.S2.V) \\
\supseteq \ \{T \text{ must be a slip of either } S1 \text{ or } S2,\text{ to which the ind. hypo. applies}\} \\
glob.\text{(slides}.T.V) \\
\]

\textit{S = “ if } B \text{ then } S1 \text{ else } S2 \text{ fi ”:}

\[
glob.(\text{slides}.“ \text{ if } B \text{ then } S1 \text{ else } S2 \text{ fi ”}.V) \\
= \ \{\text{slides of IF}\} \\
glob.“ \text{ if } B \text{ then slides}.S1.V \text{ else slides}.S2.V \text{ fi ”} \\
= \ \{\text{glob of IF}\} \\
glob.B \cup \glob.(\text{slides}.S1.V) \cup \glob.(\text{slides}.S2.V) \\
\supseteq \ \{T \text{ must be a slip of either } S1 \text{ or } S2, \text{ to which the ind. hypo. applies}\} \\
glob.(\text{slides}.T.V) \\
\]

\textit{S = “ while } B \text{ do } S1 \text{ od ”:}

\[
glob.(\text{slides}.“ \text{ while } B \text{ do } S1 \text{ od ”}.V) \\
= \ \{\text{slides of DO}\} \\
glob.“ \text{ while } B \text{ do slides}.S1.V \text{ od ”} \\
= \ \{\text{glob of DO}\} \\
glob.B \cup \glob.(\text{slides}.S1.V) \\
\supseteq \ \{T \text{ must be a slip of } S1 \text{ and the ind. hypo. applies}\} \\
glob.(\text{slides}.T.V) \\
\]

\textbf{Lemma 9.4.} Let \( S \) be a core statement; let \( T \) be any slip of \( S \); then

\[
def.T \subseteq \def.S \\
\]

\textit{Proof.} The proof is by induction over the structure of \( S \).

\( S = “ X := E “ \): Here, \( T \) must be \( X := E \), and the inclusion is trivial.

\( S = “ S1 \ ; \ S2 “ \):

\[
def.“ S1 \ ; \ S2 “ \\
= \ \{\text{def of } ' ; '\} \\
def.S1 \cup \def.S2 \\
\]

\( T \) must be a slip of either \( S1 \) or \( S2 \), to which the ind. hypo. applies. 

\( \square \)
If $T$ is either $S, S_1$ or $S_2$, the inclusion is trivial. Otherwise, it must be a slip of either $S_1$ or $S_2$, and thus $\text{def}.T$ must be included in either $\text{def}.S_1$ or $\text{def}.S_2$, respectively, due to the induction hypothesis.

$S = "\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi }":$

$\text{def." if } B \text{ then } S_1 \text{ else } S_2 \text{ fi "} = \{\text{def of IF}\}$

$\text{def}.S_1 \cup \text{def}.S_2$.

If $T$ is either $S, S_1$ or $S_2$, the inclusion is trivial. Otherwise, it must be a slip of either $S_1$ or $S_2$, and thus $\text{def}.T$ must be included in either $\text{def}.S_1$ or $\text{def}.S_2$, respectively, due to the induction hypothesis.

$S = "\text{while } B \text{ do } S_1 \text{ od }":$

$\text{def." while } B \text{ do } S_1 \text{ od "} = \{\text{def of DO}\}$

$\text{def}.S_1$.

If $T$ is either $S$ or $S_1$, the inclusion is trivial. Otherwise, it must be a slip of $S_1$, and thus $\text{def}.T$ must be included in $\text{def}.S_1$ due to the induction hypothesis.

\[\square\]

### C.2 Slide independence and liveness

**Theorem C.2.** At any program point in a slide-independent statement, any variable of the slide-independent set or one that was not defined in the original statement is alive only if it was alive in the corresponding point of the original statement. That is, let $S, XI, Y, LV$ be any statement and three sets of variables, respectively, with $XI$ slide independent in $\text{slides}.S$ (i.e. $\text{glob}.(\text{slides}.S.XI) \cap \text{def}.S \subseteq XI$) and $LV$ live-on-exit; let $LV'$ be the set of live variables at a certain point in $(\text{slides}.S.XI)[\text{live } LV]$ and let $LV''$ be the set of live variables at the corresponding point of $S[\text{live } LV]$; then $(LV' \setminus Y) \subseteq (LV'' \setminus Y)$ provided $\text{def}.S \setminus XI \subseteq Y$.

**Proof.** We prove by induction over the structure of $\text{slides}.S.XI$ a variation, stating that provided $LV_1, LV_2$ are the live variables on exit from $\text{slides}.S.XI$ and $S$, respectively, with $(LV_1 \setminus Y) \subseteq (LV_2 \setminus Y)$, we also get $(LV_1' \setminus Y) \subseteq (LV_2'' \setminus Y)$ for the sets of live variables $LV_1', LV_2'$ at any corresponding points in $(\text{slides}.S.XI)[\text{live } LV_1]$ and $S[\text{live } LV_2]$.

First, for live-on-entry variables $LV_1' := (LV_1 \setminus d\text{def}.(\text{slides}.S.XI)) \cup \text{input}.(\text{slides}.S.XI)$ and $LV_2' := (LV_2 \setminus d\text{def}.S) \cup \text{input}.S$, we observe
APPENDIX C. PROPERTIES OF SLIDES

\[ LV'_1 \backslash Y \]
\[
\{\text{def. of } LV'_1\}
((LV1 \backslash \text{ddef.}(\text{slides}.XI)) \cup \text{input.}(\text{slides}.XI)) \backslash Y
\]
\[
\{\text{set theory}\}
((LV1 \backslash ((LV1 \backslash Y) \cap \text{ddef.}(\text{slides}.XI))) \cup \text{input.}(\text{slides}.XI)) \backslash Y
\]
\[
\{\text{Lemma C.4, see below: } (LV \backslash Y) \cap \text{def.} S \subseteq XI\}
((LV1 \backslash ((LV1 \backslash Y) \cap \text{ddef.} S)) \cup \text{input.}(\text{slides}.XI)) \backslash Y
\]
\[
\{\text{set theory}\}
((LV1 \backslash \text{ddef.} S) \cup \text{input.}(\text{slides}.XI)) \backslash Y
\]
\[
\{\text{set theory: } \text{RE5 and } \text{glob.}(\text{slides}.XI) \subseteq \text{glob.} S\}
((LV1 \backslash \text{ddef.} S) \cup ((\text{glob.} S \backslash Y) \cap \text{input.}(\text{slides}.XI))) \backslash Y
\]
\[
\{\text{Lemma C.5, see below: } (\text{glob.} S \backslash Y) \cap \text{def.} S \subseteq XI\}
((LV1 \backslash \text{ddef.} S) \cup ((\text{glob.} S \backslash Y) \cap \text{input.} S)) \backslash Y
\]
\[
\{\text{set theory: } \text{input.} S \subseteq \text{glob.} S\}
((LV1 \backslash \text{ddef.} S) \cup \text{input.} S) \backslash Y
\]
\[
\{\text{set theory: proviso } (LV1 \backslash Y) \subseteq (LV2 \backslash Y)\}
((LV2 \backslash \text{ddef.} S) \cup \text{input.} S) \backslash Y
\]
\[
\{\text{set theory: } \text{input.} S \subseteq \text{glob.} S\}
LV2' \backslash Y.
\]

For internal points of \text{slides}.XI, we need to examine sequential composition, IF statements and DO loops, assuming \(XI \cap \text{def.} S \neq \emptyset\) (since otherwise we get \(\text{slides}.XI = \text{skip}\), with no internal points).

Recall that (due to Lemma [9,2]) we know that \(XI\), being slide independent in \text{slides}.S, is also slide ind. in \text{slides}.T, for any slip \(T\) of \(S\). Furthermore, since \(\text{def.} T \subseteq \text{def.} S\) for any slip \(T\) of \(S\), the proviso \(\text{def.} S \backslash XI \subseteq Y\) implies \(\text{def.} T \backslash XI \subseteq Y\). Thus, the induction hypothesis can be correctly applied to any slip, provided its respective live-on-exit variables \(LV1\) and \(LV2\) have \((LV1 \backslash Y) \subseteq (LV2 \backslash Y)\).

\(S = \text{"S1 ; S2"}\):
The induction hypothesis on \((\text{slides}.S2.XI)[\text{live} LV1]\) and \(S2[\text{live} LV2]\), ensures \((LV1' \backslash Y) \subseteq (LV2' \backslash Y)\) for any point in \text{slides}.S2.XI, including its entry. Thus the ind. hypo. for \((\text{slides}.S1.XI)[\text{live} LV1']\) and \(S1[\text{live} LV2]\) yields the requested result.
$S = \text{"if } B \text{ then } S_1 \text{ else } S_2 \text{ fi"} \):  
Here, the live-on-exit variables are also live-on-exit of both branches; the ind. hypo. then takes care of those branches.

$S = \text{"while } B \text{ do } S_1 \text{ od"} \):  
The variables live-on-exit from the loop body are exactly the variables live-on-entry (to the DO loop), $LV_1', LV_2'$; and those are already known to have the requested ($LV_1' \setminus Y \subseteq (LV_2' \setminus Y)$).

**Corollary C.3.** Slide independence preserves non-simultaneous liveness. That is, let $S, VI, X, X_1, Y$ be any statement and four sets of variables, respectively, with $VI$ slide independent in $\text{slides.}S$ (i.e. $\text{glob.} (\text{slides.}S.VI) \cap \text{def.} S \subseteq VI$) and $X$ non-simultaneously-live in $S[\text{live } X_1, Y]$; then $X$ is also non-simultaneously-live in $(\text{slides.}S.VI)[\text{live } X_1, Y]$ provided $X_1 \subseteq X, |X_1| \leq 1, Y = \text{glob.} S \setminus X$ and $X \cap \text{def.} S \subseteq VI$.

**Proof.** Let $LV'$ be the set of live variables at a certain point in $(\text{slides.}S.VI)[\text{live } X_1, Y]$; let $LV''$ be the set of live variables at the corresponding point in $S[\text{live } X_1, Y]$. From Theorem C.2 we know $(LV' \setminus Y) \subseteq (LV'' \setminus Y)$, because $VI$ is slide ind. in $\text{slides.}S$ and $\text{def.} S \setminus VI \subseteq Y$ (the latter is due to $\text{def.} S \subseteq \text{glob.} S$ and $X \cap \text{def.} S \subseteq VI$). Thus (by set theory, recall $Y = \text{glob.} S \setminus X$ and note only variables from $\text{glob.} S \cup X$ may be live) we get $X \cap LV' \subseteq X \cap LV''$. Combine this with the non-simultaneous liveness of $X$ in $S$ (i.e. $|(X \cap LV'')| \leq 1$) and the non-simultaneous liveness of $X$ in $(\text{slides.}S.VI)[\text{live } X_1, Y]$ is proved.

**Lemma C.4.** Let $S, V, X$ be any core statement and two sets of variables, respectively; then

$$(X \cap \text{ddef.} S) = (X \cap \text{ddef.} (\text{slides.}S. V))$$

provided $X \cap \text{def.} S \subseteq V$ .

**Proof.** First, for the case of $V \circ \text{def.} S$ we observe

$$X \cap \text{ddef.} (\text{slides.}S. V)$$

$$= \{\text{def. of slides when } V \circ \text{def.} S\}$$

$$X \cap \text{ddef.} \text{skip}$$

$$= \{\text{ddef of skip }\}$$

$$X \cap \emptyset$$

$$= \{\text{set theory}\}$$

$$\emptyset$$
APPENDIX C. PROPERTIES OF SLIDES

\[ \{ X \circ \text{def}.S \text{ due to } V \circ \text{def}.S \text{ and proviso; } \]
\[ \text{hence } X \circ \text{def}.S \text{ due to RE4} \}
\[ X \cap \text{def}.S . \]

When \( V \cap \text{def}.S \neq \emptyset \), we prove the inclusion by induction over the structure of \( S \).

\[ S = \text{“ } V1, Y1 := E1, E2 \text{ ” with } V1 \subseteq V \text{ and } Y1 \circ V: \]
\[ X \cap \text{def}.\langle \text{slides} \text{“ } V1, Y1 := E1, E2 \text{ ”} \rangle .V \]
\[ = \{ \text{slides of ‘:=’: } V1 \subseteq V \text{ and } Y1 \circ V \} \]
\[ X \cap \text{def} \text{“ } V1 := E1 \text{ ”} \]
\[ = \{ \text{def of ‘:=’} \} \]
\[ X \cap V1 \]
\[ = \{ \text{set theory: } X \circ Y1 \text{ since } X \cap (V1, Y1) \subseteq V \text{ and } V \circ Y1 \} \]
\[ (X \cap V1) \cup (X \cap Y1) \]
\[ = \{ \text{set theory: } V1 \circ Y1 \} \]
\[ X \cap (V1, Y1) \]
\[ = \{ \text{def of ‘:=’} \} \]
\[ X \cap \text{def} \text{“ } V1, Y1 := E1, E2 \text{ ”} . \]

\[ S = \text{“ } S1 ; S2 \text{ ”}: \]
\[ X \cap \text{def}.\langle \text{slides} \text{“ } S1 ; S2 \text{ ”} \rangle .V \]
\[ = \{ \text{slides of ‘; ’} \} \]
\[ X \cap \text{def} \text{“ } S1.V ; \text{slides}.S2.V \text{ ”} \]
\[ = \{ \text{def of ‘; ’} \} \]
\[ X \cap (\text{def}.\langle \text{slides}.S1.V \rangle \cup \text{def}.\langle \text{slides}.S2.V \rangle) \]
\[ = \{ \text{set theory} \} \]
\[ (X \cap \text{def}.\langle \text{slides}.S1.V \rangle) \cup (X \cap \text{def}.\langle \text{slides}.S2.V \rangle) \]
\[ = \{ \text{ind. hypo., twice: } \text{def}.S1 \subseteq \text{def} \text{“ } S1 ; S2 \text{ ”}; similarly for } S2 \} \]
\[ (X \cap \text{def}.S1) \cup (X \cap \text{def}.S2) \]
\[ = \{ \text{set theory} \} \]
\[ X \cap (\text{def}.S1 \cup \text{def}.S2) \]
\[ = \{ \text{def of ‘; ’} \} \]
\[ X \cap \text{def} \text{“ } S1 ; S2 \text{ ”} . \]
\[ S = \text{“ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi ”:} \]
\[ X \cap \text{ddef.(slides.“ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi ”}.V) \]
\[ = \{ \text{slides of IF} \} \]
\[ X \cap \text{ddef.“ if } B \text{ then slides.S1.V else slides.S2.V fi ”} \]
\[ = \{ \text{ddef of IF} \} \]
\[ X \cap \text{ddef.(slides.S1.V) \cap ddef.(slides.S2.V)} \]
\[ = \{ \text{set theory} \} \]
\[ (X \cap \text{ddef.(slides.S1.V)}) \cap (X \cap \text{ddef.(slides.S2.V)}) \]
\[ = \{ \text{ind. hypo., twice: ddef.S1 \subseteq ddef.IF; similarly for S2} \} \]
\[ (X \cap \text{ddef.S1}) \cap (X \cap \text{ddef.S2}) \]
\[ = \{ \text{set theory} \} \]
\[ X \cap (\text{ddef.S1 \cap ddef.S2}) \]
\[ = \{ \text{ddef of IF} \} \]
\[ X \cap \text{ddef.“ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi ”}. \]

\[ S = \text{“ while } B \text{ do } S_1 \text{ od ”:} \]
\[ X \cap \text{ddef.(slides.“ while } B \text{ do } S_1 \text{ od ”}.V) \]
\[ = \{ \text{slides of DO} \} \]
\[ X \cap \text{ddef.“ while } B \text{ do slides.S1.V od ”} \]
\[ = \{ \text{ddef of DO} \} \]
\[ X \cap \emptyset \]
\[ = \{ \text{ddef of DO} \} \]
\[ X \cap \text{ddef.“ while } B \text{ do } S_1 \text{ od ”}. \]

Lemma C.5. Let \( S, VI, X \) be any core statement and two sets of variables, respectively, with \( VI \) slide independent in \( \text{slides.S} \) (i.e. glob.(slides.S.VI) \cap \text{def}.S \subseteq VI); then
\[ (X \cap \text{input.(slides.S.VI)}) \subseteq (X \cap \text{input.S}) \]
provided \( X \cap \text{def}.S \subseteq VI \).

Proof. First, if \( VI \circ \text{def}.S \) we get \( \text{slides.S.VI} = \text{skip} \) and the inclusion becomes trivial.

When \( VI \cap \text{def}.S \neq \emptyset \), we prove the inclusion by induction over the structure of \( S \).
APPENDIX C. PROPERTIES OF SLIDES

$S = \text{"} VI1, Y1 := E1, E2 \text{"}$ with $VI1 \subseteq VI$ and $Y1 \circ VI$:

$\text{input.(slides.\text{"} VI1, Y1 := E1, E2 \text{"}.VI)}$

$= \{\text{slides of ':='}\}$

$\text{input.\text{"} VI1 := E1 \text{"}}$

$= \{\text{input of ':='}\}$

$\text{glob.E1}$

$\subseteq \{\}$

$\text{glob.E1} \cup \text{glob.E2}$

$= \{\text{input of ':='}\}$

$\text{input.\text{"} VI1, Y1 := E1, E2 \text{"}}$.

$S = \text{"} S1 ; S2 \text{"}$:

$X \cap \text{input.(slides.\text{"} S1 ; S2 \text{"}.VI)}$

$= \{\text{slides of ' ; '}\}$

$X \cap \text{input.(slides.S1.VI ; slides.S2.VI)}$

$= \{\text{input of ' ; '}\}$

$X \cap (\text{input.(slides.S1.VI)} \cup (\text{input.(slides.S2.VI) \ ddef.(slides.S1.VI)}))$

$= \{\text{set theory}\}$

$(X \cap \text{input.(slides.S1.VI)}) \cup ((X \cap \text{input.(slides.S2.VI)}) \setminus (X \cap \text{ddef.(slides.S1.VI)}))$

$\subseteq \{\text{ind. hypo., twice}\}$

$(X \cap \text{input.S1}) \cup ((X \cap \text{input.S2}) \setminus (X \cap \text{ddef.(slides.S1.VI)}))$

$= \{\text{Lemma C.4}\}$

$(X \cap \text{input.S1}) \cup ((X \cap \text{input.S2}) \setminus (X \cap \text{ddef.S1}))$

$= \{\text{set theory}\}$

$X \cap (\text{input.S1} \cup (\text{input.S2} \ ddef.S1))$

$= \{\text{input of ' ; '}\}$

$X \cap \text{input.\text{"} S1 ; S2 \text{"}}$.

$S = \text{"} \text{if } B \text{ then } S1 \text{ else } S2 \text{ fi } \text{"}$:

$X \cap \text{input.(slides.\text{"} \text{if } B \text{ then } S1 \text{ else } S2 \text{ fi } \text{.VI)}}$

$= \{\text{slides of IF}\}$

$X \cap \text{input.(if } B \text{ then slides.S1.VI else slides.S2.VI fi)}$
APPENDIX C. PROPERTIES OF SLIDES

\[ X \cap (\text{glob.} B \cup \text{input.}(\text{slides}. S1.VI)) \cup \text{input.}(\text{slides}. S2.VI) \]

\[ = \{\text{input of IF}\} \]

\[ (X \cap \text{glob.} B) \cup (X \cap \text{input.}(\text{slides}. S1.VI)) \cup (X \cap \text{input.}(\text{slides}. S2.VI)) \]

\[ \subseteq \{\text{ind. hypo., twice}\} \]

\[ (X \cap \text{glob.} B) \cup (X \cap \text{input}. S1) \cup (X \cap \text{input}. S2) \]

\[ = \{\text{set theory}\} \]

\[ X \cap (\text{glob.} B \cup \text{input}. S1 \cup \text{input}. S2) \]

\[ = \{\text{input of IF}\} \]

\[ X \cap \text{input.} “\text{ if } B \text{ then } S1 \text{ else } S2 \text{ fi }” . \]

\[ S = “\text{ while } B \text{ do } S1 \text{ od ”}: \]

\[ X \cap \text{input.} (\text{slides}. “\text{ while } B \text{ do } S1 \text{ od ”}. VI) \]

\[ = \{\text{slides of DO}\} \]

\[ X \cap \text{input.} (\text{while } B \text{ do } \text{slides}. S1.VI \text{ od}) \]

\[ = \{\text{input of DO}\} \]

\[ X \cap (\text{glob.} B \cup \text{input.}(\text{slides}. S1.VI)) \]

\[ = \{\text{set theory}\} \]

\[ (X \cap \text{glob.} B) \cup (X \cap \text{input.}(\text{slides}. S1.VI)) \]

\[ \subseteq \{\text{ind. hypo.}\} \]

\[ (X \cap \text{glob.} B) \cup (X \cap \text{input}. S1) \]

\[ = \{\text{set theory}\} \]

\[ X \cap (\text{glob.} B \cup \text{input}. S1) \]

\[ = \{\text{input of DO}\} \]

\[ X \cap \text{input.} “\text{ while } B \text{ do } S1 \text{ od ”} . \]

\[ \square \]
Appendix D

SSA

D.1 General derivation

The transformation to and from SSA will be based on the following general derivation.

Program equivalence D.1. A variable in $X$ may or may not be live-on-exit; independently, it may or may not be live-on-entry and it may be self-defined or normally defined or not at all defined. So potentially we have 12 cases. However, in our context, some combinations are not possible. Firstly, if a variable is self-defined, the used instances must be live-on-entry. Secondly, a variable may not be live-on-exit-only, unless it is actually defined. So we are left with nine cases to be distinguished.

For self-definitions, we have variables live-on-entry-only ($XL_1^i := XL_1$) or live-on-both ($XL_2 := XL_2i$); for normally defined variables, we have the live-on-both, live-on-entry-only, live-on-exit-only and the dead variables, respectively ($XL_3^i, XL_4, XL_5^i, XL_6 := E_1', E_2', E_3', E_4'$); of the non-defined variables, we have variables live-on-both ($X_7$), live-on-entry-only ($X_8$) and, again, dead variables ($X_9$).

We note that subsets $X_1, X_2, X_3, X_4, X_7, X_8$ are live-on-entry, with initial instances $XL_1i, XL_2i, XL_3i, XL_4i, XL_7i, XL_8i$, respectively. The final instances $XL_1f, XL_3f, XL_5f, XL_7i$ of subsets $X_1, X_3, X_5, X_7$ are all live-on-exit. Finally, note that subsets $XL_2, XL_4, XL_6$ represent dead assignments.

Let $XLs$ be the set of all instances; let $Y, Y_1$ be two more sets of program variables with $Y_1 \subseteq Y$ and $Y$ live on exit; finally, let $E_1, E_2, E_3, E_4, E_5, E_1', E_2', E_3', E_4', E_5'$ be ten lists of
expressions; then

\[
\begin{align*}
& \quad \text{“ (X}_3, X_4, X_5, X_6, Y_1 := E_1, E_2, E_3, E_4, E_5 ; \\
& \quad \quad XL_1 f, XL_3 f, XL_5 f, XL_7 i := X_1, X_3, X_5, X_7) [\text{live } XL_1 f, XL_3 f, XL_5 f, XL_7 i, Y] ” = \\
& \quad \text{“ (XL}_1 i, XL_2 i, XL_3 i, XL_4 i, XL_7 i, XL_8 i := X_1, X_2, X_3, X_4, X_7, X_8 ; \\
& \quad \quad XL_1 f, XL_2, XL_3 f, XL_4, XL_5 f, XL_6, Y_1 := XL_1 i, XL_2 i, E_1^\prime, E_2^\prime, E_3^\prime, E_4^\prime, E_5^\prime) \\
& \quad \quad [\text{live } XL_1 f, XL_3 f, XL_5 f, XL_7 i, Y] ” \\
& \text{provided}
\end{align*}
\]

P1: \quad (X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8) \subseteq X,

P2: \quad (XL_1 i, XL_2 i, XL_3 i, XL_4 i, XL_7 i, XL_8 i) \subseteq XLs,

P3: \quad (XL_1 f, XL_2, XL_3 f, XL_4, XL_5 f, XL_6, XL_7 i) \subseteq XLs,

P4: \quad Y_1 \subseteq Y,

P5: \quad (X, XLs, Y) \text{ disjoint},

P6: \quad \text{glob.(E}_1, E_2, E_3, E_4, E_5) \subseteq (X_1, X_2, X_3, X_4, X_7, X_8, Y),

P7: \quad [E_1^\prime = E_1[X_1, X_2, X_3, X_4, X_7, X_8 \setminus XL_1 i, XL_2 i, XL_3 i, XL_4 i, XL_7 i, XL_8 i]],

P8: \quad [E_2^\prime = E_2[X_1, X_2, X_3, X_4, X_7, X_8 \setminus XL_1 i, XL_2 i, XL_3 i, XL_4 i, XL_7 i, XL_8 i]],

P9: \quad [E_3^\prime = E_3[X_1, X_2, X_3, X_4, X_7, X_8 \setminus XL_1 i, XL_2 i, XL_3 i, XL_4 i, XL_7 i, XL_8 i]],

P10: \quad [E_4^\prime = E_4[X_1, X_2, X_3, X_4, X_7, X_8 \setminus XL_1 i, XL_2 i, XL_3 i, XL_4 i, XL_7 i, XL_8 i]] \text{ and }

P11: \quad [E_5^\prime = E_5[X_1, X_2, X_3, X_4, X_7, X_8 \setminus XL_1 i, XL_2 i, XL_3 i, XL_4 i, XL_7 i, XL_8 i]].

\text{Proof.}\n
\[
\begin{align*}
& \quad \text{“ (XL}_1 i, XL_2 i, XL_3 i, XL_4 i, XL_7 i, XL_8 i := X_1, X_2, X_3, X_4, X_7, X_8 ; \\
& \quad \quad XL_1 f, XL_2, XL_3 f, XL_4, XL_5 f, XL_6, Y_1 := XL_1 i, XL_2 i, E_1^\prime, E_2^\prime, E_3^\prime, E_4^\prime, E_5^\prime) \\
& \quad \quad [\text{live } XL_1 f, XL_3 f, XL_5 f, XL_7 i, Y] ” = \\
& \quad \{\text{assignment-based sub. (Law 18)}\} \text{ due to P1,P2 and P5} \\
& \quad (X_1, X_2, X_3, X_4, X_7, X_8) \circ (XL_1 i, XL_2 i, XL_3 i, XL_4 i, XL_7 i, XL_8 i); \\
& \quad \text{remove redundant double-sub.: P7-P11 and then P1,P2,P5,P6 give} \\
& \quad (XL_1 i, XL_2 i, XL_3 i, XL_4 i, XL_7 i, XL_8 i) \circ \text{glob.}(E_1, E_2, E_3, E_4, E_5) \\
& \quad \text{“ (XL}_1 i, XL_2 i, XL_3 i, XL_4 i, XL_7 i, XL_8 i := X_1, X_2, X_3, X_4, X_7, X_8 ; \\
& \quad \quad XL_1 f, XL_2, XL_3 f, XL_4, XL_5 f, XL_6, Y_1 := X_1, X_2, E_1, E_2, E_3, E_4, E_5) \\
& \quad \quad [\text{live } XL_1 f, XL_3 f, XL_5 f, XL_7 i, Y] ” = \\
& \quad \{\text{remove dead assignment (Law 23)}\} \text{ due to P1,P2,P5 and P6} \\
& \quad (XL_1 i, XL_2 i, XL_3 i, XL_4 i, XL_8 i) \circ \\
& \quad \quad (((XL_7 i, Y) \setminus Y_1) \cup \text{glob.}(E_1, E_2, E_3, E_4, E_5)) \}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{“ (XL}_7 i := X_7 ; XL_1 f, XL_2, XL_3 f, XL_4, XL_5 f, XL_6, Y_1 := \\
& \quad \quad X_1, X_2, E_1, E_2, E_3, E_4, E_5)[\text{live } XL_1 f, XL_3 f, XL_5 f, XL_7 i, Y] ”
\end{align*}
\]
APPENDIX D. SSA

\[\{\text{intro. dead assignment (Law 23): due to P1,P2 and P5} \}
\]
\[
(X3, X4, X5, X6) \odot (XL1f, XL3f, XL5f, XL7i, Y) \]

\[
\{\text{intro. following assertion (Laws 7, 8)}\}
\[
(XL7i := X7 ; XL1f, XL2, XL3f, X3, XL4, X4, XL5f, X5, XL6, X6, Y1 := \]
\[
X1, X2, E1, E2, E3, E4, E5[\text{ live } XL1f, XL3f, XL5f, XL7i, Y] \}
\]

\[\{\text{intro. following assignment (Law 6)}\}
\]
\[
(XL7i := X7 ; XL1f, XL2, XL3f, X3, XL4, X4, XL5f, X5, XL6, X6, Y1 := \]
\[
X1, X2, E1, E2, E3, E4, E5 ; \{X1, X3, X5 = XL1f, XL3f, XL5f\}
\]
\[
) \{\text{live } XL1f, XL3f, XL5f, XL7i, Y] \}
\]

\[\{\text{remove following assertion (Laws 7, 8)}\}
\]
\[
(XL7i := X7 ; XL1f, XL2, XL3f, X3, XL4, X4, XL5f, X5, XL6, X6, Y1 := \]
\[
X1, X2, E1, E2, E3, E4, E5 ;
\]
\[
) \{\text{live } XL1f, XL3f, XL5f, XL7i, Y] \}
\]

\[\{\text{remove dead assignment (Law 23: due to P1,P3 and P5} \}
\]
\[
(XL1f, XL2, XL3f, XL4f, XL5f, XL6) \odot (XL7i, Y, X1, X3, X5) \]

\[\{\text{swap statements (Law 5.1: } X7 \odot (X3, X4, X5, X6, Y1), \text{ due to P1,P4 and P5; and} \]
\[
XL7i \odot ((X3, X4, X5, X6, Y1) \cup \text{glob.}(E1, E2, E3, E4, E5)), \text{ by P1,P2,P4,P5} \}
\]
\[
(X3, X4, X5, X6, Y1 := E1, E2, E3, E4, E5 ;
\]
\[
) \{\text{live } XL1f, XL3f, XL5f, XL7i, Y] \}
\]

\[\{\text{merge assignments (Law 1: } XL7i \odot (X1, X3, X5), \text{ by P1,P2,P5} \}
\]
\[
(X3, X4, X5, X6, Y1 := E1, E2, E3, E4, E5 ;
\]
\[
) \{\text{live } XL1f, XL3f, XL5f, XL7i, Y] \}
\]

Program equivalence D.2. Let \(S1, S2, S1', S2', X1, X2, XL1i, XL2i, Y\) be four statements and
five sets of variables, respectively; then
\[
(S1 ; S2 ; XL2f := X2)[\text{live } XL2f, Y] \] = \( (XL1i := X1 ; S1' ; S2')[\text{live } XL2f, Y] \)
provided

P1: “(S1 ; XL3 := X3)[live XL3, Y]” = “(XL1i := X1 ; S1')[live XL3, Y]” and

P2: “(S2 ; XL2f := X2)[live XL2f, Y]” = “(XL3 := X3 ; S2')[live XL2f, Y]”

where XL3 := ((XL2f \ ddef.S2') ∪ (input.S2') \ Y).

Proof.

“(XL1i := X1 ; S1' ; S2')[live XL2f, Y]”

= {prop. liveness info.: by def. of X3 (and set theory) we get

(XL3, Y) ⊃ (((XL2f, Y) \ ddef.S2') ∪ input.S2')}“((XL1i := X1 ; S1')[live XL3, Y] ; S2')[live XL2f, Y]”

= {P1}“((S1 ; XL3 := X3)[live XL3, Y] ; S2')[live XL2f, Y]”

= {remove liveness info.: again (XL3, Y) ⊃ (((XL2f, Y) \ ddef.S2') ∪ input.S2')}“(S1 ; XL3 := X3 ; S2')[live XL2f, Y]”

= {prop. liveness info.}“(S1 ; (XL3 := X3 ; S2')[live XL2f, Y])[live XL2f, Y]”

= {P2}“(S1 ; (S2 ; XL2f := X2)[live XL2f, Y])[live XL2f, Y]”

= {remove liveness info.}“(S1 ; S2 ; XL2f := X2)[live XL2f, Y]”.


\[\square\]

Program equivalence D.3. Let B, B', S1, S2, S1', S2', X1, X2, XL1i, XL2i, Y be two boolean expressions, four statements and five sets of variables, respectively; then

“(if B then S1 else S2 fi ; XL2f := X2)[live XL2f, Y]” = “(XL1i := X1 ; if B' then S1' else S2' fi)[live XL2f, Y]”

provided

P1: [B' ≡ B[X1 \ XL1i]],
P2: XL1i ⊆ X1,
P3: XL1i ⊆ glob.B,
P4: “(S1 ; XL2f := X2)[live XL2f, Y]” = “(XL1i := X1 ; S1')[live XL2f, Y]” and

P5: “(S2 ; XL2f := X2)[live XL2f, Y]” = “(XL1i := X1 ; S2')[live XL2f, Y]”.
Proof.

“(XL1i := X1 ; if B′ then S1′ else S2′)\b]|\text{live } XL2f, Y]”
\begin{align*}
\text{=} & \quad \{P1\} \\
\text{=} & \quad \{\text{assignment-based sub. (Law 18): } XL1i \text{ } \circ \text{ } X1 \text{ (P2)}\} \\
\text{=} & \quad \{\text{prop. liveness info.}\} \\
\text{=} & \quad \{\text{dist. IF over ‘; ’ (Law 4)}\} \\
\end{align*}

\textbf{Program equivalence D.4.} Let B, B′, S1, S1′, X1, X2, XL1i, XL2i, Y be two boolean expressions, two statements and five (disjoint) sets of variables; then

“(DO ; XL2i := X2)[\text{live } XL2i, Y]” = “(XL1i, XL2i := X1, X2 ; DO′)[\text{live } XL2i, Y]”
where $DO := "\text{ while } B \text{ do } S_1 \text{ od }"$ and $DO' := "\text{ while } B' \text{ do } S'_1 \text{ od }"

provided

P1: $(X_1, X_2, XL_1i, XL_2i, Y)$ are disjoint,
P2: $(XL_1i, XL_2i) \circ \text{ input}.DO$,
P3: $\text{input}.DO' \subseteq (XL_1i, XL_2i, Y)$,
P4: $[B' \equiv B[X_1, X_2 \setminus XL_1i, XL_2i]]$ and

P5: $\begin{array}{l}
\quad (S_1 ; XL_1i, XL_2i := X_1, X_2)[\text{live } XL_1i, XL_2i, Y]
\quad = \\
\quad (XL_1i, XL_2i := X_1, X_2 ; S'_1)[\text{live } XL_1i, XL_2i, Y]
\end{array}$

{
Proof.

\[
\begin{array}{l}
\quad (XL_1i, XL_2i := X_1, X_2 ; \text{while } B' \text{ do } S'_1 \text{ od})[\text{live } XL_2i, Y]
\quad = \\
\quad \text{intro. dead assignment (Law 26), due to P1,P3}
\quad (X_1, X_2) \circ ((XL_2i, Y) \cup \text{glob.B'} \cup \text{input.S1'})
\quad (XL_1i, XL_2i := X_1, X_2 \text{ while } B' \text{ do } S'_1 ; X_1, X_2 := XL_1i, XL_2i \text{ od})[\text{live } XL_2i, Y] \quad \\
\quad \text{intro. following assertion (Law 7), twice}
\quad (XL_1i, XL_2i := X_1, X_2 ; \{XL_1i, XL_2i = X_1, X_2\} \text{ while } B' \text{ do } S'_1 ; X_1, X_2 := XL_1i, XL_2i \text{ od})[\text{live } XL_2i, Y] \quad \\
\quad \text{prop. assertion (Law 13)}
\quad (XL_1i, XL_2i := X_1, X_2 ; \{XL_1i, XL_2i = X_1, X_2\} \text{ while } B' \text{ do } S'_1 ; X_1, X_2 := XL_1i, XL_2i \text{ od})[\text{live } XL_2i, Y] \quad \\
\quad \text{intro. following assignment (Law 6)}
\quad (XL_1i, XL_2i := X_1, X_2 ; \{XL_1i, XL_2i = X_1, X_2\} \text{ while } B' \text{ do } \{XL_1i, XL_2i = X_1, X_2\} \text{ od})[\text{live } XL_2i, Y] \quad \\
\quad \text{prop. assertion (Law 13)}
\quad (XL_1i, XL_2i := X_1, X_2 ; \{XL_1i, XL_2i = X_1, X_2\} \text{ while } B' \text{ do } XL_1i, XL_2i := X_1, X_2 ; S'_1 ; X_1, X_2 := XL_1i, XL_2i \text{ od})[\text{live } XL_2i, Y] \quad \\
\quad \text{remove following assertion (Law 7), twice}
\quad (XL_1i, XL_2i := X_1, X_2 ; \{XL_1i, XL_2i = X_1, X_2\} \text{ while } B' \text{ do } XL_1i, XL_2i := X_1, X_2 ; S'_1 ; X_1, X_2 := XL_1i, XL_2i \text{ od})[\text{live } XL_2i, Y]
\end{array}
\]
\[
\begin{align*}
  &= \{ \text{liveness analysis: P3 gives } \text{input.DO}' \subseteq (XL_1i, XL_2i, Y) \} \\
  \text{“(XL}_1i, XL_2i := X_1, X_2 \text{; while } B' \text{ do} \\
  ((XL}_1i, XL_2i := X_1, X_2 ; S'_1)[\text{live XL}_1i, XL_2i, Y] ; \\
  X_1, X_2 := XL_1i, XL_2i][\text{live XL}_1i, XL_2i, Y] \text{ od}[\text{live XL}_2i, Y] \}
\end{align*}
\]

\[
\begin{align*}
  &= \{ \text{P5} \} \\
  \text{“(XL}_1i, XL_2i := X_1, X_2 \text{; while } B' \text{ do} \\
  (S_1 ; XL_1i, XL_2i := X_1, X_2)[\text{live XL}_1i, XL_2i, Y] ; \\
  X_1, X_2 := XL_1i, XL_2i][\text{live XL}_1i, XL_2i, Y] \text{ od}[\text{live XL}_2i, Y] \}
\end{align*}
\]

\[
\begin{align*}
  &= \{ \text{remove liveness info.} \} \\
  \text{“(XL}_1i, XL_2i := X_1, X_2 \text{; while } B' \text{ do} \\
  (S_1 ; XL_1i, XL_2i := X_1, X_2)[\text{live XL}_1i, XL_2i, Y] \text{ od}[\text{live XL}_2i, Y] \}
\end{align*}
\]

\[
\begin{align*}
  &= \{ \text{assignment-based sub. (Law 18: } (X_1, X_2) \circ (XL_1i, XL_2i) \text{ due to P1} \} \\
  \text{“(XL}_1i, XL_2i := X_1, X_2 \text{; while } B' \text{ do} \\
  (S_1 ; XL_1i, XL_2i := X_1, X_2)[\text{live XL}_1i, XL_2i, Y] \text{ od}[\text{live XL}_2i, Y] \}
\end{align*}
\]

\[
\begin{align*}
  &= \{ \text{remove redundant self-assignment (Law 2: remove liveness info.} \} \\
  \text{“(XL}_1i, XL_2i := X_1, X_2 \text{; while } B' \text{ do } S_1 ; XL_1i, XL_2i := X_1, X_2 \text{ od}[\text{live XL}_2i, Y] \}
\end{align*}
\]

\[
\begin{align*}
  &= \{ \text{P4} \} \\
  \text{“(XL}_1i, XL_2i := X_1, X_2 \text{; while } B[X_1, X_2 \setminus XL_1i, XL_2i] \text{ do} \\
  S_1 ; XL_1i, XL_2i := X_1, X_2 \text{ od}[\text{live XL}_2i, Y] \}
\end{align*}
\]

\[
\begin{align*}
  &= \{ \text{assignment-based sub. (Law 18: } (X_1, X_2) \circ (XL_1i, XL_2i) \} \\
  \text{“(XL}_1i, XL_2i := X_1, X_2 \text{; while } B[X_1, X_2 \setminus XL_1i, XL_2i][XL_1i, XL_2i \setminus X_1, X_2] \text{ do} \\
  S_1 ; XL_1i, XL_2i := X_1, X_2 \text{ od}[\text{live XL}_2i, Y] \}
\end{align*}
\]

\[
\begin{align*}
  &= \{ \text{remove redundant (reversed) double-sub.: P2 gives } (XL_1i, XL_2i) \circ \text{glob.B} \} \\
  \text{“(XL}_1i, XL_2i := X_1, X_2 \text{; while } B \text{ do } S_1 ; XL_1i, XL_2i := X_1, X_2 \text{ od}[\text{live XL}_2i, Y] \}
\end{align*}
\]

\[
\begin{align*}
  &= \{ \text{code motion (Law 5: due to P1,P2} \\
  \text{(XL}_1i, XL_2i) \circ (\text{glob.B} \cup \text{input.S1} \cup (X_1, X_2)) \}
\end{align*}
\]

\[
\begin{align*}
  &= \{ \text{remove dead assignment (Law 23: by P1 we get } XL_1i \circ (XL_2i, Y) \} \\
  \text{“(XL}_1i, XL_2i := X_1, X_2 \text{; while } B \text{ do } S_1 \text{ od ; XL}_1i, XL_2i := X_1, X_2[\text{live XL}_2i, Y] \}
\end{align*}
\]

\[
\begin{align*}
  &= \{ \text{remove dead assignment (Law 23: by P1 we get } XL_1i \circ (XL_2i, Y) \} \\
  \text{“(XL}_1i, XL_2i := X_1, X_2 \text{; while } B \text{ do } S_1 \text{ od ; XL}_2i := X_2[\text{live XL}_2i, Y] \}
\end{align*}
\]
D.2 Transform to SSA

We now apply the results of the above general derivation in deriving an algorithm to transform any given program statement to SSA.

Transformation D.5. Let $S$, $X$, $Y$ be any core statement and two (disjoint) sets of variables; let $X_1, X_2, X_3, X_4, X_5$ be five (mutually disjoint) subsets of $X$, and let $XL_1i, XL_2i, XL_3i, XL_4i, XL_5f$ be six sets of instances, all included in the full set of instances $XLs$; let $S'$ be the SSA form of $S$ defined by

$$S' := \text{toSSA}(S, X, (XL_1i, XL_2i, XL_3i, XL_4i), (XL_3i, XL_4f, XL_5f), Y, XLs);$$

then (Q1:)

“(while $B$ do $S1$ od ; $XL_2i := X2$)[live $XL_2i, Y$]”

and (Q2:)

“( $S$ ; $XL_3i, XL_4f, XL_5f := X3, X4, X5$)[live $XL_3i, XL_4f, XL_5f, Y$]”

and (Q2:) $X \diamond \text{glob}.S'$ provided

P1: $\text{glob}.S \subseteq (X, Y)$,

P2: $(X_1, X_2, X_3, X_4, X_5) \subseteq X$,

P3: $(XL_1i, XL_2i, XL_3i, XL_4i, XL_4f, XL_5f) \subseteq XLs$,

P4: $XLs \diamond (X, Y)$,

P5: $(X_1, X_3) \diamond \text{def}.S$,

P6: $(X_2, X_4, X_5) \subseteq \text{def}.S$ and

P7: $(X \cap (((X_3, X_4, X_5) \setminus \text{ddef}.S) \cup \text{input}.S)) \subseteq (X_1, X_2, X_3, X_4)$.

Preconditions P1 and P2 identifies all program variables (in $S$); then P3 and P4 (along with P1) ensure all instances $XLs$ are fresh; P5 and P6 (along with P3 and the repetition of $XL_2i$ in Q1) ensure any live-on-exit final instance is also live-on-entry if and only if its respective program variable is not defined in $S$ (if it is both defined and live-on-entry, a different initial instance will be used); finally, P7 (along with P2) makes postcondition Q2 achievable, by demanding the availability of an initial instance to all live-on-entry variables (in $X$).
Proof. The derivation of the toSSA algorithm is given hand in hand with its proof of correctness. For a given statement $S$, we assume for any slip $T$ of $S$ the correctness of its toSSA transformation (provided all preconditions are met) in proving the correctness for $S$ itself.

As will be seen, in the course of the following derivation, for each case of $S' := \text{toSSA}(S, X, (XL_1i, XL_2i, XL_3i, XL_4i), (XL_3i, XL_4f, XL_5f), Y, XLs)$ we shall be obliged to show

$$\text{DP1: } ((XL_3i, XL_4f, XL_5f) \setminus \text{ddef}.S') \cup (\text{input}.S' \setminus Y)) \subseteq (XL_1i, XL_2i, XL_3i, XL_4i)$$

and

$$\text{DP2: } (XL_4f, XL_5f) \subseteq \text{ddef}.S'$$

provided P1-P7 hold.

In order to allow all recursive calls to introduce fresh instances, without clashing with surrounding variables, we shall make the names of all global program variables (i.e. from $X$ and $Y$) invariably available to recursive calls. Since those calls will be applied to slips of $S$, P1 will be guaranteed (due to $\text{glob}.T \subseteq \text{glob}.S$ for any slip $T$ of $S$).

Furthermore, whenever fresh instances are introduced, they will be added to $XLs$ in further recursive calls. Thus the inclusion of all instances in $XLs$, as required by P3, will be maintained. This way, choosing all fresh names to be distinct from $(X, Y, XLs)$ will maintain P4. For keeping P5,P6 and the disjointness of instances $(XL_1i, XL_2i, XL_3i, XL_4i, XL_4f, XL_5f)$ (as implied by P3), special care will be needed. In the various cases, such considerations will be key in deriving the details of the transformation. Finally, for P7 to be invariably maintained, we shall propagate liveness information backward (following our laws of liveness analysis) whilst propagating forward assignments to initial instances (following both laws of program manipulation and postcondition Q1).

Assignment

We begin by analyzing the relevant subsets of live variables $X_1$-$X_5$; of those, variables $X_2, X_4, X_5$ are defined in $S$, with $X_2, X_4$ live-on-entry and $X_4, X_5$ live-on-exit. Furthermore, there is a potential of defined variables $X_6 \subseteq (X \setminus (X_1, X_2, X_3, X_4, X_5))$; those are neither live-on-entry nor live-on-exit (i.e. dead assignments, as with $X_2$).

All remaining defined variables $(Y_1 \circ X)$ will have to be from $Y$ (due to P1).

For variables $(X_3, X_4)$, being live on both entry and exit, we recall from Q1 and P5 the non-defined $X_3$ must have the same initial and final instances $(XL_3i)$, whereas (from Q1,P3 and P6) the defined $X_4$ should be given fresh initial instances $XL_4i$.

Likewise, all dead assignments to $(X_2, X_6)$ must yield fresh instances, $(XL_2, XL_6)$. (An alternative would have been to allow the merging algorithm to remove dead assignments. This would have caused complications in changing the results of liveness analysis and thus raise questions — which are better avoided — over the order of translation. Instead, one can perform the elimination
of dead assignments independently of the merging.)

In terms of Program equivalence \[D.1\] we observe (for Q1) that our \(X4, X2, X5, X6, X3, X1\) correspond to \(X3, X4, X5, X6, X7, X8\) over there. Thus

\[
\begin{align*}
\text{“}&(X4, X2, X5, X6, Y1 := E1, E2, E3, E4, E5 ; \\
&Xl3i, XlAf, Xl5f := X3, X4, X5)[\text{live } Xl3i, XlAf, Xl5f, Y1] \text{ ”} \\
= &\{\text{Program equivalence } D.1 \text{ with } \\
&X1, X2, X3, X4, X5, X6, X7, X8 := \emptyset, \emptyset, X4, X2, X5, X6, X3, X1, \\
&Xl3i, XlAi, Xl7i, Xl8i := XlAi, Xl2i, Xl3i, Xl1i, \\
&Xl3f, XlAf, Xl5f, Xl6f := XlAf, Xl2, Xl5f, Xl6, \\
&Xls := (Xls, Xl2, Xl4i, Xl6) \text{ and } \\
&(E1’, E2’, E3’, E4’, E5’) := (E1, E2, E3, E4, E5) \\
&[X1, X2, X3, X4 \setminus Xl1i, Xl2i, Xl3i, Xl4i]: \\
P1 \text{ is due to our P2; P2 and P3 are due to our P3 and the fresh choices; } \\
P4 \text{ holds by choice of } Y1 \odot X \text{ and our P1; } \\
P5 \text{ is a result of our P4 and the fresh choices; } \\
P6 \text{ is due to our P1,P2 and P7; finally P7-P11 hold by construction}\}
\end{align*}
\]

\[
\text{“}&(Xl1i, Xl2i, Xl3i, Xl4i := X1, X2, X3, X4 ; \\
&XlAf, Xl2, Xl5f, Xl6, Y1 := E1’, E2’, E3’, E4’, E5’) \\
&[\text{live } Xl3i, XlAf, Xl5f, Y1] \text{ ” .}
\]

We thus derive \(\text{toSSA.(“ } X4, X2, X5, X6, Y1 := E1, E2, E3, E4, E5 \text{ ”) }\),

\[
X, (Xl1i, Xl2i, Xl3i, Xl4i), (Xl3i, XlAf, Xl5f), Y, Xls)) \Leftrightarrow \\
“ XlAf, Xl2, Xl5f, Xl6, Y1 := E1’, E2’, E3’, E4’, E5’ ”
\]

where \((E1’, E2’, E3’, E4’, E5’, E6’) := (E1, E2, E3, E4, E5, E6)\)

\[
[X1, X2, X3, X4 \setminus Xl1i, Xl2i, Xl3i, Xl4i]
\]

and \((Xl2, Xl6) := \text{fresh.}((X2, X6), (X, Y, Xls)) \) .

Q2. Indeed \(X \odot \text{glob.} “ XlAf, Xl2, Xl5f, Xl6, Y11, Y2 := E1’, E2’, E3’, E4’, E5’, E6’ ” \) since \((X \cap \text{glob.}(E1, E2, E3, E4, E5, E6)) \subseteq (X1, X2, X3, X4) \) (due to input of ‘:=’ and P7) and

\((X1, X2, X3, X4) \) are all substituted by elements of \(Xls\) (P3) which are disjoint from \(X\) (P4).

DP1.

\[
((Xl3i, XlAf, Xl5f) \setminus (XlAf, Xl2, Xl5f, Xl6, Y1)) \cup \\
(glob.(E1’, E2’, E3’, E4’, E5’, E6’) \setminus Y)
\]
The key here is to determine an intermediate set of instances for which all preconditions (P1-P7) hold for the two recursive calls to toSSA. Let \((X_1, X_2, X_3, X_4)\) be the live-on-entry variables, and let \((X_3, X_4, X_5)\) be live-on-exit, with \((X_1, X_3) \circ (\text{def}. S_1 \cup \text{def}. S_2)\) and \((X_2, X_4, X_5) \subseteq (\text{def}. S_1 \cup \text{def}. S_2)\). We have no problem with \(X_3\) as all instances (i.e. final, intermediate and initial) will have to be the same \((XL3i)\) since those are not defined in \(\text{S} 1 \; \text{S} 2\). The remaining live intermediate variables are \(X_6 := ((X \setminus X_3) \cap (((X_4, X_5) \setminus \text{def}. S_2) \cup \text{input}. S_2))\).

Of \(X_6\), variables in \(X_{11}, X_{21}, X_{41} := (X_1 \cap X_6), ((X_2 \cap X_6) \setminus \text{def}. S_1), ((X_4 \cap X_6) \setminus \text{def}. S_1)\) will have to reuse the initial instances \(XL1i, XL2i, XL4i\). Similarly, variables in \(X_{42}, X_{51} := (((X_4 \cap X_6) \setminus \text{def}. S_2), (((X_5 \cap X_6) \setminus \text{def}. S_2)\) will reuse final instances \(XL4f, XL5f\). (Note that \(X_{41} \circ X_{42}\) since due to \(X_4 \subseteq (\text{def}. S_1 \cup \text{def}. S_2)\) — variables in \(X_{41}\) must be in \(\text{def}. S_2\) whereas variables in \(X_{42}\) must not.)

Finally, the remaining variables \(X_{61} := (X_6 \setminus (X_{11}, X_{21}, X_{41}, X_{42}, X_{51}))\) must get fresh instances \(XL61 := \text{fresh}.(X_{61}, (X, Y, XLs))\). In summary, the intermediately-live instances will be

\[ XL6 := (XL1i, XL2i, XL3i, XL4i, XL4f, XL5f, XL61). \]

The above construction of \(XL6\), along with the given P1-P7 and the associativity of liveness analysis (Lemma 7.2 for P7) ensure

\[ S1' := \text{toSSA}.(S1, X, (XL1i, XL2i, XL3i, XL4i), XL6, Y, XLs') \quad \text{— where } XLs' := (XLs, XL61) \quad \text{enjoys all its P1-P7 and thus (Q1)} \]

\[ "(S1 \; XL6 := X6)[live XL6, Y]" = \]

\[ "(XL1i, XL2i, XL3i, XL4i := X1, X2, X3, X4 \; S1' \; S2')[live XL6, Y]" . \]

Similarly, all P1-P7 for \(S2' := \text{toSSA}.(S2, X, XL6, (XL3i, XL4f, XL5f), Y, XLs'')\) are guaranteed for any \(XLs'' \supset XLs'\), thus yielding (Q1)
APPENDIX D. SSA

“ (S2 ; XL3i, XLAF, XL5f := X3, X4, X5)[live XL3i, XLAF, XL5f, Y] ” =
“ (XL6 := X6 ; S2’)[live XL3i, XLAF, XL5f, Y] ”.

In toSSA, we shall insist on XLs” := (XLs’ ∪ (glob.S1’ \ Y)) in order to avoid defining the same
instance twice, and thus losing the static-single-assignment property.

Q1.

“ (S1 ; S2 ; XL3i, XLAF, XL5f := X3, X4, X5)[live XL3i, XLAF, XL5f, Y1] ”
= (Program equivalence \[2\] with (XL6, XLs’, XLs”) as defined above)
XL1i := (XL1i, XL2i, XL3i, XL4i); XL2f := (XL3i, XLAF, XL5f);
S1’ := toSSA(S1, X, (XL1i, XL2i, XL3i, XL4i), XL6, Y, XLs’);
S2’ := toSSA(S2, X, XL6, (XL3i, XLAF, XL5f), Y, XLs”);
ind. hypo. (Q1), twice: P1-P7 of both cases justified above

“ (XL1i, XL2i, XL3i, XL4i := X1, X2, X3, X4 ; S1’ ; S2’)
[live XL3i, XLAF, XL5f, Y1] ”.

We thus derive
toSSA.(“ S1 ; S2 ”, X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XLAF, XL5f), Y, XLs) \[\triangleq \]
“ S1’ ; S2’ ”

where X6 := ((X \ X3) ∩ ((X4, X5) \ ddef.S2) \ input.S2)),
X11 := (X1 \ X6),
X21 := ((X2 \ X6) \ def.S1),
X41 := ((X4 \ X6) \ def.S1),
X42 := ((X4 \ X6) \ def.S2),
X51 := ((X5 \ X6) \ def.S2),
X61 := (X6 \ (X11, X21, X41, X42, X51)),
XL61 := fresh(X61, (X, Y, XLs)),
XL6 := (XL11i, XL21i, XL3i, XL41i, XL42f, XL51f, XL61),
XLs’ := (XLs, XL61),
S1’ := toSSA.(S1, X, (XL1i, XL2i, XL3i, XL4i), XL6, Y, XLs’),
XLs” := (XLs’ ∪ (glob.S1’ \ Y))
and S2’ := toSSA.(S2, X, XL6, (XL3i, XLAF, XL5f), Y, XLs”) .

Q2. Indeed X \ glob.“ S1’ ; S2’ ” due to the ind. hypo. (Q2), twice.

DP1.

\[((XL3i, XLAF, XL5f) \ ddef.“ S1’ ; S2’ ”) \cup ((input.“ S1’ ; S2’ ”) \ Y)\]
= \{associativity of liveness (Lemma 7.2)\}
\(((((XL3i, XLAF, XL5f) \ ddef.S2’) \cup input.S2’) \cup ddef.S1’) \cup input.S1’ \ Y)\)
APPENDIX D. SSA

\[ \text{set theory} \]
\[
(((XL3i, XLAf, XL5f) \setminus \text{ddef.S2'}) \cup \text{(input.S2') \setminus \text{ddef.S1'}}) \cup \text{input.S1'} \setminus Y
\]
\[ \subseteq \text{ind. hypo. (DP1 of S2')} \]
\[
((XL6 \setminus \text{ddef.S1'}) \cup \text{input.S1'}) \setminus Y
\]
\[ = \text{set theory: } XL6 \circ Y \text{ by construction of XL6 (P3,P4 and freshness of XL61)} \]
\[
(XL6 \setminus \text{ddef.S1'}) \cup \text{(input.S1'} \setminus Y)
\]
\[ \subseteq \text{ind. hypo. (DP1 of S1')} \]
\[
(XL4i, XL2i, XL3i, XL4i)
\]

DP2. Due to ddef of ‘;’, we need to show \((XLAf, XL5f) \subseteq (\text{ddef.S1'} \cup \text{ddef.S2'})\). Final instances of variables in \((X4, X5) \cap \text{def.S2}\) are in ddef.S2’ due to the ind. hypo. (i.e. DP2 of S2’). The remaining elements of \((X4, X5)\) must be in \(X6 \setminus \text{def.S2}\) and hence in \((X41, X51)\). Thus, their final instances must be in \((XL41f, XL51f)\) and hence in ddef.S1’ due to the ind. hypo. (i.e. DP2 of S1’) as required.

**IF**

The key this time lies in DP2 and the definition of ddef of IF. Variables in \((X4, X5)\) are defined in the IF statement and must be defined in both branches of the resulting IF’. We achieve that by ending both the then and else branches of IF’ with assignments to the final instances \((XLAf, XL5f)\).

But what do we assign to members of \((XLAf, XL5f)\)? In each branch the answer will be different, depending on whether the variable is defined in that branch and if not, whether it is live on entry (i.e. in XL4i) or not.

Variables in \(X4d1 := (X4 \cap (\text{def.S1} \setminus \text{def.S2}))\) should be given fresh instances \((XL4d1t)\) in the then branch but use their initial instances \((XL4d1i)\) as final instances in the else branch. (Failing to reuse such initial instances would inevitably render DP2 false, by yielding simultaneous liveness.)

Similarly, variables in \(X4d2 := (X4 \cap (\text{def.S2} \setminus \text{def.S1}))\) should be given fresh instances \((XL4d2e)\) in the else branch but reuse initial instances \(XL4d2i\) as final instances in the then branch.

Now each of the remaining variables of \(X4\) (i.e. in \(X4d1d2 := X4 \setminus (X4d1 \cup X4d2)\)) and each member of X5 should be given two (distinct) fresh instances (i.e. \((XL4d1d2, XL4d1d2e, XL5t, XL5e)\)). Those new instances will in turn act as final instances in the two recursive calls.

Finally, for brevity, we define XL4t := (XL4d1t, XL4d2i, XL4d1d2t) and XL4e := (XL4d1i, XL4d2e, XL4d1d2e). In summary, the then branch will end with an assignment “XL4f, XL5f := XL4t, XL5t ”. Similarly, the else branch will end with the assignment
APPENDIX D. SSA

“\(XLAf, XL5f := XL4e, XL5e\).”

The above construction along with the given P1-P7 ensure
\(S1' := \text{toSSA}(S1, X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XL4t, XL5t), Y, XLs')\) — where \(XLs' := (XLs, XL4d1t, XL4d2e, XL4d1d2t, XL4d1d2e, XL5t, XL5e)\) — enjoys all its P1-P7. Similarly, all P1-P7 for
\(S2' := \text{toSSA}(S2, X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XL4e, XL5e), Y, XLs'')\) are guaranteed for any \(XLs'' \supseteq XLs'\). As before, in order to avoid double assignments to any instance, we shall insist on \(XLs'' := (XLs' \cup (\text{glob}, S1' \setminus Y))\).

We now aim to apply Program equivalence \([1, 3]\) with \(S1'' := “S1' ; XLAf, XL5f := XL4t, XL5t”\) and \(S2'' := “S2' ; XLAf, XL5f := XL4e, XL5e”\). For this to be correct, we have to show

“\((S1 ; XL3i, XLAf, XL5f := X3, X4, X5)[live XL3i, XLAf, XL5f, Y1]\)” =
“\((XL1i, XL2i, XL3i, XL4i := X1, X2, X3, X4 ; S1'')\)[live XL3i, XLAf, XL5f, Y1]” and
“\((S2 ; XL3i, XLAf, XL5f := X3, X4, X5)[live XL3i, XLAf, XL5f, Y]\)” =
“\((XL1i, XL2i, XL3i := X1, X2, X3 ; S2'')[live XL3i, XLAf, XL5f, Y]\).” For the former, we observe

“\((XL1i, XL2i, XL3i, XL4i := X1, X2, X3, X4 ; S1'')\)[live XL3i, XLAf, XL5f, Y]\)”
= \{def. of \(S1''\)\}

“\((XL1i, XL2i, XL3i, XL4i := X1, X2, X3, X4 ; S1' ;
XLAf, XL5f := XL4t, XL5t)[live XL3i, XLAf, XL5f, Y]\)”
= \{prop. liveness\}

“\(((XL1i, XL2i, XL3i, XL4i := X1, X2, X3, X4 ; S1')\)[live XL3i, XL4t, XL5t, Y] ;
XLAf, XL5f := XL4t, XL5t)[live XL3i, XLAf, XL5f, Y]\)”
= \{ind. hypo. (Q1 of \(S1'\))\}

“\(((S1 ; XL3i, XL4t, XL5t := X3, X4, X5)[live XL3i, XL4t, XL5t, Y] ;
XLAf, XL5f := XL4t, XL5t)[live XL3i, XLAf, XL5f, Y]\)”
= \{remove liveness\}

“\((S1 ; XL3i, XL4t, XL5t := X3, X4, X5 ;
XLAf, XL5f := XL4t, XL5t)[live XL3i, XLAf, XL5f, Y]\)”
= \{assignment-based sub. (Law 18): \((X4, X5) \circ (XL3i, XL4t, XL5t)\) due to P2,P3,P4 and the freshness of \((XL4t, XL5t)\)\}

“\((S1 ; XL3i, XL4t, XL5t := X3, X4, X5 ;
XLAf, XL5f := X4, X5)[live XL3i, XLAf, XL5f, Y]\)”
= \{ \text{remove dead assignments: } (XL4t, XL5t) \circ (XL3i, X4, X5, Y) \text{ again, by } \\
P2, P3, P4 \text{ and the freshness of } (XL4t, XL5t) \} \\
\text{“ } (S1 \.; XL3i \; ::= \; X3 \.; XLAf, XL5f \; ::= \; X4, X5)\text{[live XL3i, XLAf, XL5f, Y] ” }
\]
= \{ \text{merge following assignments (Law 1): } XL3i \circ (XLAf, XL5f, X4, X5) \text{ by P2, P3, P4} \} \\
\text{“ } (S1 \.; XL3i, XLAf, XL5f \; ::= \; X3, X4, X5)\text{[live XL3i, XLAf, XL5f, Y] }”. \\

The corresponding proof for $S2''$ is similar (and thus omitted). We are now ready to transform the IF statement into IF’, as follows:

\[ \text{“ } (\text{if } B \text{ then } S1 \text{ else } S2 \text{ if } ; \\
XL3i, XLAf, XL5f \; ::= \; X3, X4, X5)\text{[live XL3i, XLAf, XL5f, Y] ” } \]
\]
\[ = \text{[Program equivalence D.3 with } S1”, S2” \text{ as defined above, } \\
B' \; ::= \; B[X1, X2, X3, X4 \setminus XL1i, XL2i, XL3i, XL4i], \\
XL1i \; ::= \; (XL1i, XL2i, XL3i, XL4i) \text{ and } XL2f \; ::= \; (XL3i, XLAf, XL5f): } \\
P1 \text{ holds by construction (of } B'); P2 \text{ holds due to our P2, P3, P4; } \\
P3 \text{ holds due to P1, P3, P4; and P4, P5 hold as proved above} \\
\text{“ } (XL1i, XL2i, XL3i, XL4i \; ::= \; X1, X2, X3, X4 ; \\
\text{if } B' \text{ then } S1' \.; XLAf, XL5f \; ::= \; XLAt, XL5t \\
\text{else } S2' \.; XLAf, XL5f \; ::= \; XL4e, XL5e \text{ if } \text{[live XL3i, XLAf, XL5f, Y] ” } \].

We thus derive

tossa.(IF, X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XLAf, XL5f), Y, XLs) \triangleq \text{IF’}

where IF := “ if B then S1 else S2 if ”,

\[ \text{IF’ := “ } \text{ if } B[X1, X2, X3, X4 \setminus XL1i, XL2i, XL3i, XL4i] \\
\text{then } S1' \.; XLAf, XL5f \; ::= \; XL4t, XL5t \text{ else } S2' \.; XLAf, XL5f \; ::= \; XL4e, XL5e \text{ if } ”, \\
X4d1 := (X4 \cap (\text{def.S1} \setminus \text{def.S2})), \\
X4d2 := (X4 \cap (\text{def.S2} \setminus \text{def.S1})), \\
X4d1d2 := X4 \cap \text{def.S1} \cap \text{def.S2}, \\
(XL4d1t, XL4d1e, XL4d1d2t, XL4d1d2e, XL5t, XL5e) := \\
fresh.((X4d1, X4d2, X4d1d2, X4d1d2e, X5, X5), (X, Y, XLs)), \\
XL4t := (XL4d1t, XL4d2i, XL4d1d2t), \\
XL4e := (XL4d1i, XL4d2e, XL4d1d2e), \\
XLs' := (XLs, XL4d1t, XL4d2e, XL4d1d2t, XL4d1d2e, XL5t, XL5e), \\
S1' := tossa.(S1, X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XL4t, XL5t), Y, XLs'), \\
XLs'' := (XLs' \cup (\text{glob.}S1' \setminus Y)) \\
\text{and } S2' := tossa.(S2, X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XL4e, XL5e), Y, XLs'') \].
Q2. Indeed $X \circ \text{glob}.IF'$ due to the ind. hypo. (Q2 of $S'1$ and $S'2$) and since $(X \cap \text{glob}.B) \subseteq (X1, X2, X3, X4)$ by P7.

DP1.

$\{(XL3i, XL4f, XL5f) \setminus \text{ddef}.IF'\} \cup (\text{input}.IF' \setminus Y)$

$= \{\text{set theory: } (XL4f, XL5f) \subseteq \text{ddef}.IF'$ and $XL3i \circ \text{ddef}.IF'\} \cup (\text{input}.IF' \setminus Y)$

$= \{\text{set theory: } (XL4f, XL5f) \subseteq \text{ddef}.IF' \}$

$XL3i \cup (\text{input}.IF' \setminus Y)$

$= \{\text{set theory: } (XL4f, XL5f) \subseteq \text{ddef}.IF' \}$

$= \{\text{input of } IF'\}$

$XL3i \cup ((\text{glob}.B' \cup \text{input} \cdot S'1 \cup (\text{input} \cdot S'2 \cup (XL4f, XL5f := XL4t, XL5t "\cup \text{input} \cdot S'2 \cup (XL4f, XL5f := XL4e, XL5e) ")) \setminus Y)$

$= \{\text{ind. hypo., twice: } (XL4d1t, XL4d1d2t, XL5t) \subseteq \text{ddef}.S'1' (\text{DP2 of } S'1') \text{ and } (XL4d2e, XL4d12d2e, XL5e) \subseteq \text{ddef}.S'2' (\text{DP2 of } S'2')\}

$$\subseteq \{\text{set theory: } (XL4d1i, XL4d2i) \subseteq XL4i\}

(XL3i, XL4i) \cup ((\text{glob}.B' \cup \text{input}.S'1' \cup \text{input}.S'2') \setminus Y)

= \{\text{set theory\}}

(XL3i, XL4i) \cup (\text{glob}.B' \setminus Y) \cup (\text{input}.S'1' \setminus Y) \cup (\text{input}.S'2' \setminus Y)

$\subseteq \{\text{def. of } B' \text{ and } \text{glob}.B \subseteq (X1, X2, X3, X4, Y) \text{ due to P1 and P7}\}$

$\subseteq \{\text{ind. hypo., twice: } (\text{input}.S'1' \setminus Y) \subseteq (XL1i, XL2i, XL3i, XL4i) \text{ (DP1 of } S'1') \text{ and } (\text{input}.S'2' \setminus Y) \subseteq (XL1i, XL2i, XL3i, XL4i) \text{ (DP1 of } S'2')\}$

$\subseteq \{\text{ind. hypo., twice: } (\text{input}.S'1' \setminus Y) \subseteq (XL1i, XL2i, XL3i, XL4i) \text{ (DP1 of } S'1') \text{ and } (\text{input}.S'2' \setminus Y) \subseteq (XL1i, XL2i, XL3i, XL4i) \text{ (DP1 of } S'2')\}$

DP2. By construction, we indeed have $(XL4f, XL5f) \subseteq \text{ddef}.IF'$.

**DO**

We need to enforce the policy of non-simultaneous liveness of the instances of a program variable. Since $\text{ddef}.DO$ is empty, the final instance of a live-on-exit variable must also be live-on-entry to the loop. Thus, the set of live-on-exit-only variables $X5$ is empty. Furthermore, if the (live-on-exit) variable is also in $\text{input}.DO$, it must be the final instance that is live-on-entry to the SSA.
loop $DO'$. We achieve that by defining such instances that are also defined in the loop ($XLAf$) just before the loop begins and at the end of its body. Similarly, other variables in $\text{def.DO} \cap \text{input.DO}$ (not being live-on-exit) should have a dedicated (fresh) loop-entry instance ($XL2$). Thus, when recursively transforming the loop body $S1$ to SSA, non-simultaneous liveness is ensured by sending instances ($XL1i, XL2, XL3i, XLAf$) as initial values. Now what about final values for $S1'$?

Non-defined live variables ($X1, X3$) should be given the corresponding initial instances ($XL1i, XL3i$). As for the defined variables ($X2, X4$), fresh instances ($XL2b, XL4b$) must be invented for their final $S1'$ value. These should in turn be assigned back to the loop-entry instances, just after $S1'$.

The above construction along with the given P1-P7 ensure

$S1' := \text{toSSA}(S1, X, (XL1i, XL2, XL3i, XLAf), (XL1i, XL2b, XL3i, XL4b), Y, XLs')$

— where $XLs' := (XLs, XL2, XL2b, XL4b) —$ enjoys all its P1-P7.

We now aim to apply Program equivalence $\text{def.DO}$ with $S1' := \langle S1' ; XL2, XLAf := XL2b, XL4b \rangle$ for $S1'$, $B[X1, X2, X3, X4 \setminus XL1i, XL2, XL3i, XLAf]$ for $B'$ and $(XL1i, XL2), (XL3i, XL4f)$ for $XL1i, XL2i$ respectively. For this to be correct, we have to show its P1-P5. P1 is a result of our P1-P4 and the freshness of $XL2$; P2 is given by our P1,P3,P4 along with RE5 (which yields $\text{input.DO} \subseteq \text{glob.DO}$); P3 is due to the ind. hypo. (DP1 of $S1'$) and the def. of $B'$ along with P1,P7 (yielding $\text{glob.B} \subseteq (X1, X2, X3, X4, Y)$); P4 is correct by construction (of $B'$); and finally, for P5, we observe

“ (S1 ; XL1i, XL2, XL3i, XLAf := X1, X2, X3, X4)[live XL1i, XL2, XL3i, XLAf, Y] ” =

“ (XL1i, XL2, XL3i, XLAf := X1, X2, X3, X4 ; S1')[live XL1i, XL2, XL3i, XLAf, Y] ”:

“ (XL1i, XL2, XL3i, XLAf := X1, X2, X3, X4 ; S1')[live XL1i, XL2, XL3i, XLAf, Y] ”

= {def. of $S1'$}

“ (XL1i, XL2, XL3i, XLAf := X1, X2, X3, X4 ; S1' ; XL2, XLAf := XL2b, XL4b)[live XL1i, XL2, XL3i, XLAf, Y] ”

= {prop. liveness}

“ ((XL1i, XL2, XL3i, XLAf := X1, X2, X3, X4 ; S1')[live XL1i, XL2, XL3i, XLAf, Y] ;
XL2, XLAf := XL2b, XL4b)[live XL1i, XL2, XL3i, XLAf, Y] ”

= {ind. hypo. (Q1 of $S1'$)}

“ ((S1 ; XL1i, XL2, XL3i, XLAf := X1, X2, X3, X4)
[live XL1i, XL2, XL3i, XLAf, Y] ;
XL2, XLAf := XL2b, XL4b)[live XL1i, XL2, XL3i, XLAf, Y] ”
We thus derive

\[
\begin{align*}
\text{DO} :&= \{\text{remove liveness}\} \\
\text{do } \prime &\in \text{Law 18}: (X_2, X_4) \circ (XL_1i, XL_2, XL_3i, XLAf) \\
\text{DO} &\equiv (XL_1i, XL_2, XL_3i, XLAf := X_1, X_2, X_3, X_4) \\
\text{while } &\text{live } XL_1i, XL_2, XL_3i, XLAf, Y \\
\text{do } \prime &\in \text{Law 18}: (X_2, X_4) \circ (XL_1i, XL_2, XL_3i, XLAf) \\
\text{DO} &\equiv (XL_1i, XL_2, XL_3i, XLAf := X_1, X_2, X_3, X_4) \\
\text{while } &\text{live } XL_1i, XL_2, XL_3i, XLAf, Y \\
\text{do } \prime &\in \text{Law 18}: (X_2, X_4) \circ (XL_1i, XL_2, XL_3i, XLAf) \\
\text{DO} &\equiv (XL_1i, XL_2, XL_3i, XLAf := X_1, X_2, X_3, X_4) \\
\text{while } &\text{live } XL_1i, XL_2, XL_3i, XLAf, Y \\
\text{do } \prime &\in \text{Law 18}: (X_2, X_4) \circ (XL_1i, XL_2, XL_3i, XLAf) \\
\text{DO} &\equiv (XL_1i, XL_2, XL_3i, XLAf := X_1, X_2, X_3, X_4) \\
\text{while } &\text{live } XL_1i, XL_2, XL_3i, XLAf, Y \\
\text{do } \prime &\in \text{Law 18}: (X_2, X_4) \circ (XL_1i, XL_2, XL_3i, XLAf) \\
\text{DO} &\equiv (XL_1i, XL_2, XL_3i, XLAf := X_1, X_2, X_3, X_4) \\
\text{while } &\text{live } XL_1i, XL_2, XL_3i, XLAf, Y \\
\end{align*}
\]

Let \( \text{DO} : \text{"} S_1 \text{"} \) and

\( \text{DO'} : \text{"} S_1' ; XL_2, XLAf := XL_2b, XL_4b \text{"} \). We are now ready to transform the

\( \text{DO} \) statement, as follows:

\[
\begin{align*}
\text{DO} :&= \{\text{Program equivalence \[4.4\] as explained and justified above}\} \\
\text{DO} &\equiv (XL_1i, XL_2, XL_3i, XLAf := X_1, X_2, X_3, X_4) \\
\text{while } &\text{live } XL_1i, XL_2, XL_3i, XLAf, Y \\
\text{DO} &\equiv (XL_1i, XL_2, XL_3i, XLAf := X_1, X_2, X_3, X_4) \\
\text{while } &\text{live } XL_1i, XL_2, XL_3i, XLAf, Y \\
\text{DO} &\equiv (XL_1i, XL_2, XL_3i, XLAf := X_1, X_2, X_3, X_4) \\
\text{while } &\text{live } XL_1i, XL_2, XL_3i, XLAf, Y \\
\text{DO} &\equiv (XL_1i, XL_2, XL_3i, XLAf := X_1, X_2, X_3, X_4) \\
\text{while } &\text{live } XL_1i, XL_2, XL_3i, XLAf, Y \\
\end{align*}
\]

We thus derive

\( \text{toSSA}(\text{DO}, X, (XL_1i, XL_2i, XL_3i, XL_4i), (XL_3i, XLAf), Y, XLs) \equiv \)
APPENDIX D. SSA

“XL2, XLAf := XL2i, XLAi ; DO’”

where DO := “while B do S1 od”,

DO’ := “while B’ do S1’ ; XL2, XLAf := XL2b, XLAb od”

(XL2, XL2b, XLAb) := fresh((X2, X2, X4), (X, Y, XLs)),

XLs’ := (XLs, XL2, XL2b, XLAb),

B’ := B|X1, X2, X3, X4 \ XL1i, XL2, XL3i, XLAf|

and S1’ := toSSA(S1, X, (XL1i, XL2, XL3i, XLAf), (XL1i, XL2b, XL3i, XLAb), Y, XLs’). Q2. Indeed X ∩ glob.“XL2, XLAf := XL2i, XLAi ; DO’” due to the ind. hypo. (Q2 of S1’)

and since P7 gives (X ∩ glob.B) ⊆ (X1, X2, X3, X4).

DP1.

((XL3i, XLAf) \ ddef.“XL2, XLAf := XL2i, XLAi ; DO’”) ∪
(input.“XL2, XLAf := XL2i, XLAi ; DO’” \ Y)

= {ddef and input of ‘:=’, ‘;’ and DO}

((XL3i, XLAf) \ (XL2, XLAf)) ∪ ((XL2i, XL4i) ∪ (input.DO’ \ (XL2, XLAf))) \ Y

= {set theory: XL3i ∩ (XL2, XLAf) due to P3 and the freshness of XL2;

(XL2i, XLAi) \ Y due to P3,P4}

(XL2i, XL3i, XLAi) \ ((input.DO’ \ (XL2, XLAf)) \ Y)

= {input of DO’ and set theory}

(XL2i, XL3i, XLAi) \ ((glob.B’ \ input.“S1’ ; XL2, XLAf := XL2b, XLAb ”) \ (XL2, XLAf, Y))

⊆ {set theory: (glob.B’ \ (XL2, XLAf, Y)) ⊆ (XL1i, XL3i)

due to P1,P6 and X ∩ DO’}

(XL1i, XL2i, XL3i, XLAi) \ 
(input.“S1’ ; XL2, XLAf := XL2b, XLAb ” \ (XL2, XLAf, Y))

= {input of ‘;’ ‘;’ and ‘:=’}

(XL1i, XL2i, XL3i, XLAi) \ ((input.S1’ \ ((XL2b, XLAb) \ ddef.S1’)) \ (XL2, XLAf, Y))

⊆ {ind. hypo. (DP1 of S1’): (input.S1’ \ Y) ⊆ (XL1i, XL2, XL3i, XLAf)}

(XL1i, XL2i, XL3i, XLAi) \ (XL2b, XLAb) \ ddef.S1’ (derived property DP2):

(XL2b, XLAb) are final instances in S1’, of defined variables (X2, X4) ⊆ def.S1

(XL1i, XL2i, XL3i, XLAi).

DP2.
For variables in $X4$ we observe that the corresponding final instances $XLAf$ are clearly in $ddef$.“ $XL2, XLAf := XL2i, XLAi$ ” and hence in $ddef$.“ $XL2, XLAf := XL2i, XLAi ; DO'$ ” as required.

D.3 Back from SSA

The following is a derivation of $S := \text{fromSSA.}(S', X, XL1i, XL2f, Y, XLs)$ when $S'$ includes at most one live instance of any variable in $X$ at each program point. The goal is to turn

“ $(XL1i := X1 ; S')[live XL2f, Y]$ ” with $X \diamond glob. S'$ into the equivalent

“ $(S ; XL2f := X2)[live XL2f, Y]$ ” with $XLs \diamond glob. S$. This way, a program statement in SSA form can be turned back to the original, as in the following derivation:

```
| \[
| [(\text{var } XLi, XLf, XLim} ; XL1i := X1 ; S' ; X := XLf)]
| = \{\text{def. of live with } Y := def.S' \setminus XLs\}
| " (XL1i := X1 ; S' ; X := XLf)[live X, Y] "
| = \{\text{prop. liveness}\}
| " ((XL1i := X1 ; S')[live XLf, Y] ; X := XLf)[live X, Y] "
| = \{S := \text{fromSSA.}(S, X, XL1i, XLf, Y, XLs) (Transformation D.6, see below)\}
| " ((S ; XLf := X)[live XLf, Y] ; X := XLf)[live X, Y] "
| = \{\text{remove aux. liveness info.}\}
| " (S ; XLf := X ; X := XLf)[live X, Y] "
| = \{\text{assignment-based sub.; remove self-assignment;}\n| \text{remove dead assignment; remove aux. liveness info.}\}
| S .
```

Instead of deriving the $\text{fromSSA}$ algorithm directly from the SSA form, we shall take a more general approach. The goal is to allow some transformations (e.g. slicing) to be performed on the SSA form itself, before returning to the original form.

We observe that the return from SSA involves the merge of all instances (in $XLs$) back into the original program variables ($X$). We hypothesize that as long as there is no simultaneous liveness of any two instances (to be merged) at any program point, and as long as no such instances are simultaneously defined (i.e. in a statement of multiple assignment), the merge of all instances should be possible. Insisting on removal of self-assignments (after, or while merging) will allow assignments to pseudo instances (in IFs and DO loops) to be eliminated.
Thus we shall develop an algorithm for merging sets of variables and use it for returning from SSA, as follows:
\[ \text{fromSSA}(S', X, XL1i, XLf, Y, XLs) \equiv \text{merge-vars}(S', XLs, X, XL1i, XLf, Y) \]

**Transformation D.6.** Let \( S' \) be any core statement and \( (XL1i \cup XL2f) \subseteq XLs \); let \( S \) be a statement defined by
\[
S := \text{merge-vars}(S', XLs, X, XL1i, XL2f, Y);
\]
then (Q1:)
\[
" (XL1i := X1; S')[\text{live } XL2f, Y] " = " (S; XL2f := X2)[\text{live } XL2f, Y] "
\]
and (Q2:)
\[
XLs \circ \text{glob.S}
\]
provided
\[
P1: \quad \text{glob.S'} \subseteq (XLs, Y),
\]
\[
P2: \quad (XL1i \cup XL2f) \subseteq XLs,
\]
\[
P3: \quad (X1 \cup X2) \subseteq X,
\]
\[
P4: \quad X \circ (XLs, Y),
\]
\[
P5: \quad \text{no two instances of any member of } X \text{ are sim.-live at any point in } S'[\text{live } XL2f, Y],
\]
\[
P6: \quad (XLs \cap ((XL2f \setminus ddef.S') \cup \text{input.S'})) \subseteq XL1i,
\]
\[
P7: \quad \text{no def-on-live: i.e. no instance is defined where another instance is live-on-exit},
\]
\[
P8: \quad \text{no multiple-defs: i.e. each assignment defines at most one instance (of any } X.i).}
\]

**Proof.** As with the toSSA algorithm, the derivation of merge-vars (and hence of fromSSA) is given hand in hand with its proof of correctness. For a given statement \( S' \), we assume for any slip \( T' \) of \( S' \) the correctness of its merge-vars transformation in proving the correctness for \( S' \) itself.

**Assignment**

Aiming to use Program equivalence\[D.1\] we distinguish eight disjoint subsets of \( X \) \((X1-X8)\) and the corresponding defined instances \((XL1f, XL2, XL3f, XL4f, XL5f, XL6)\) (indeed at most one of each is defined, due to our P8), of which \( XL1f, XL3f, XL5f \) are live-on-exit whereas \( XL2, XL4, XL6 \) represent dead assignments) and used instances \((XL1i, XL2i, XL3i, XL4i, XL7i, XL8i)\), of which \( XL7i \) is live-on-both entry and exit, and the rest are live-on-entry only).

In terms of our expression of merge-vars and its preconditions P1-P8, we rewrite its \( X1 \) as \((X1, X2, X3, X4, X7, X8)\), \( XL1i \) as \((XL1i, XL2i, XL3i, XL4i, XL7i, XL8i)\), \( X2 \) as \((X1, X3, X5, X7)\) and \( XL2f \) as \((XL1f, XL3f, XL5f, XL7i)\). According to this decomposition, our target Q1 can be
rewritten as

\[
\text{“ (} (X_{L1}i, X_{L2}i, X_{L3}i, X_{L4}i, X_{L7}i, X_{L8}i := X_1, X_2, X_3, X_4, X_7, X_8 ; S') \\
[\text{live } X_{L1}f, X_{L3}f, X_{L5}f, X_{L7}i, Y] \text{” }
\]

\[
= \text{“ (} (S ; X_{L1}f, X_{L3}f, X_{L5}f, X_{L7}i := X_1, X_3, X_7) \\
[\text{live } X_{L1}f, X_{L3}f, X_{L5}f, X_{L7}i, Y] \text{” }
\]

with \( S' \) being the assignment

\[ X_{L1}f, X_{L2}, X_{L3}f, X_{L4}, X_{L5}f, X_{L6}, Y_1 := X_{L1}i, X_{L2}i, E_{1'}, E_{2'}, E_{3'}, E_{4'}, E_{5'} \]. Accordingly, pre-
conditions P1-P8 can be understood as

P1: glob.\( S' \) \( \subseteq (X_{Ls}, Y) \),

P2: \( ((X_{L1}i, X_{L2}i, X_{L3}i, X_{L4}i, X_{L7}i, X_{L8}i) \cup (X_{L1}f, X_{L3}f, X_{L5}f, X_{L7}i)) \subseteq X_{Ls} \),

P3: \( ((X_1, X_2, X_3, X_4, X_7, X_8) \cup (X_1, X_3, X_5, X_7)) \subseteq X \),

P4: \( X \circ (X_{Ls}, Y) \),

P5: no two instances of any member of \( X \) are simultaneously live at any point in \( S' \),

P6: \( (X_{Ls} \cap (X_{L7}i \cup \text{input} \cdot S')) \subseteq (X_{L1}i, X_{L2}i, X_{L3}i, X_{L4}i, X_{L7}i, X_{L8}i) \),

P7: no def-on-live: \( i.e. (X_2, X_4, X_6) \circ (X_1, X_3, X_5, X_7) \),

P8: no multiple-defs: \( i.e. (X_1, X_2, X_3, X_4, X_5, X_6) \) are disjoint.

For Program equivalence \( \square \) to be correct, we need to verify its P1-P11.

P1 ((\( X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8) \subseteq X \)) holds by construction;

P2 ((\( X_{L1}i, X_{L2}i, X_{L3}i, X_{L4}i, X_{L7}i, X_{L8}i) \subseteq X_{Ls} \)) is due to our P2;

P3 ((\( X_{L1}f, X_{L2}, X_{L3}f, X_{L4}, X_{L5}f, X_{L6}, X_{L7}i) \subseteq X_{Ls} \)) is due to our P2, P7 and the disjointness of instances \( (X_{Ls}.i \circ X_{Ls}.j \text{ for all } i \neq j) \);

P4 (\( Y_1 \subseteq Y \)) is due to our P1;

P5 (disjointness of \( (X, X_{Ls}, Y) \)) is due to our P4;

for P6-P11 to hold, we define \( E_1, E_2, E_3, E_4, E_5 := (E_{1'}, E_{2'}, E_{3'}, E_{4'}, E_{5'})[X_{L1}i, X_{L2}i, X_{L3}i, X_{L4}i, X_{L7}i, X_{L8}i \setminus X_1, X_2, X_3, X_4, X_7, X_8]; \) now P6 \((\text{glob.}(E_1, E_2, E_3, E_4, E_5) \subseteq (X_1, X_2, X_3, X_4, X_7, X_8, Y)) \) is due to our P1, P6 and the redundancy of reversed double sub. \(( (X_1, X_2, X_3, X_4, X_7, X_8) \circ \text{glob.}(E_{1'}, E_{2'}, E_{3'}, E_{4'}, E_{5'}) \text{ due to P1,P4); the latter argument proves P7-P11 as well.\)

We are now ready to apply Program equivalence \( \square \) as follows:

\[
\text{“ (} (X_{L1}i, X_{L2}i, X_{L3}i, X_{L4}i, X_{L7}i, X_{L8}i := X_1, X_2, X_3, X_4, X_7, X_8 ; \\
X_{L1}f, X_{L2}, X_{L3}f, X_{L4}, X_{L5}f, X_{L6}, Y_1 := X_{L1}i, X_{L2}i, E_{1'}, E_{2'}, E_{3'}, E_{4'}, E_{5'}) \\
[\text{live } X_{L1}f, X_{L3}f, X_{L5}f, X_{L7}i, Y] \text{” }
\]

\[
= \{ \text{Program equivalence } \square \text{ with } \\
E_1, E_2, E_3, E_4, E_5 := (E_{1'}, E_{2'}, E_{3'}, E_{4'}, E_{5'}) \\
[XL_{1}i, XL_{2}i, XL_{3}i, XL_{4}i, XL_{7}i, XL_{8}i \setminus X_{1}, X_{2}, X_{3}, X_{4}, X_{7}, X_{8}] \} \]
“(X3, X4, X5, X6, Y1 := E1, E2, E3, E4, E5;
XL1f, XL3f, XL5f, XL7i := X1, X3, X5, X7)[live XL1f, XL3f, XL5f, XL7i, Y]”.

We thus derive merge-vars.“(XL1f, XL2, XL3f, XL4, XL5f, XL6, Y1 := XL1i, XL2i, E1’, E2’, E3’, E4’, E5’),
XLs, X, (XL1i, XL2i, XL3i, XL4i, XL7i, XL8i), (XL1f, XL3f, XL5f, XL7i), Y) ≜
“X3, X4, X5, X6, Y1 := E1, E2, E3, E4, E5”

where E1, E2, E3, E4, E5 := (E1’, E2’, E3’, E4’, E5’)
[XL1i, XL2i, XL3i, XL4i, XL7i, XL8i \ X1, X2, X3, X4, X7, X8].

Q2. XLs ◦ glob.“X3, X4, X5, X6, Y1 := E1, E2, E3, E4, E5”
since (due to P2) (XLs ◦ glob.(E1’, E2’, E3’, E4’, E5’)) ⊆ (XL1i, XL2i, XL3i, XL4i, XL7i, XL8i)
and (XL1i, XL2i, XL3i, XL4i, XL7i, XL8i) ◦ (X1, X2, X3, X4, X7, X8) (due to P2-P4).

Sequential composition

We define XL3 := (XLs \ ((XL2f \ ddef.S2’) ◦ input.S2’)) to be the set of intermediately-live instances. Due to P5 (no simultaneously-live instances), XL3 includes at most one instance of each member of X, as required for the two recursive calls: S1 := merge-vars.(S1’, XLs, X, XL1i, XL3, Y)
and S2 := merge-vars.(S2’, XLs, X, XL3, XL2f, Y). Preconditions P1, P4, P5, P7 and P8 are trivial consequences of S1’ and S2’ both being slips of S’ and following our construction of S1 and S2 (in terms of the parameters to merge-vars). P2 for both cases is fine as well, due to our P2 and the construction of XL3. P3 is fine due to our P3 and by construction of X3, the set of program variables (in X) corresponding to XL3. Finally, P6 for the call to S2’ is trivial, again by construction of XL3; it is also correct for S1’, due to our P6 and the associativity of liveness (Lemma 7.2).

As a result, we get
“(S2 ; XL2f := X2)[live XL2f, Y]” = “(XL3 := X3 ; S2’)[live XL2f, Y]”
and
“(S1 ; XL3 := X3)[live XL3, Y]” = “(XL1i := X1 ; S1’)[live XL3, Y]”, as required (as P1 and P2 respectively) in the following:

“(XL1i := X1 ; S1’ ; S2’)[live XL2f, Y]”

= {Program equivalence D.2 with
S1 := merge-vars.(S1’, XLs, X, XL1i, XL3, Y);
S2 := merge-vars.(S2’, XLs, X, XL3, XL2f, Y)}

“(S1 ; S2 ; XL2f := X2)[live XL2f, Y]”.

We thus derive merge-vars.“(S1’ ; S2’), XLs, X, XL1i, XL2f, Y) ≜
“merge-vars.(S1’, XLs, X, XL1i, XL3, Y) ; merge-vars.(S2’, XLs, X, XL3, XL2f, Y)”
where $XL3 := XLs \cap ((XL2f \setminus ddef.S2') \cup input.S2')$.

Q2. As required, $XLs \circ glob. " S1 ; S2 "$ due to the ind. hypo. (Q2 of S1 and S2).

**IF**

All live-on-exit variables are also live-on-exit to each branch. Similarly, it should be safe to assume all IF’s live-on-entry variables are live-on-entry to each branch. This may be an over-approximation, but a harmless one, since the initial instances of all those variables are, in any case, available on entry to both branches. This harmfulness can be verified by the observations that $ddef$ of each branch is a superset of $ddef.IF'$ and that the corresponding $input$ is a subset of $input.IF'$.

Thus, the recursive calls, computing $S1 := merge-vars.(S1', XLs, X, XL1i, XL2f, Y)$ and $S2 := merge-vars.(S2', XLs, X, XL1i, XL2f, Y)$, faithfully maintain P6. Since the calls are to slips of $IF'$, and since all remaining parameters are identical, all other preconditions P1-P5,P7,P8 (to both recursive calls) trivially hold. As a result, we get

" $(S1 ; XL2f := X2)[live XL2f, Y]$ " = " $(XL1i := X1 ; S1')[live XL2f, Y]$ " and

" $(S2 ; XL2f := X2)[live XL2f, Y]$ " = " $(XL1i := X1 ; S2')[live XL2f, Y]$ ", as required (in P4 and P5 respectively) for

" $(XL1i := X1 ; if B' then S1' else S2' fi)[live XL2f, Y]$ "

= 

{Program equivalence \textbf{D.3} with

$B := B'[XL1i \setminus X1]$;

$S1 := merge-vars.(S1', XLs, X, XL1i, XL2f, Y)$;

$S2 := merge-vars.(S2', XLs, X, XL1i, XL2f, Y)$:

P1 ($[B' \equiv B'[XL1i \setminus X1][X1 \setminus XL1i]]$) is due to $XL1 \circ glob.B'$ (our P1,P3,P4) and the redundancy of reversed double sub.;

P2 ($XL1i \circ X1$) is due to our P2-P4; it also proves

P3 ($XL1i \circ glob.B'[XL1i \setminus X1]$); finally

the ind. hypo. (Q1), twice, give P4 and P5

" $(if B then S1 else S2 fi ; XL2f := X2)[live XL2f, Y]$ " .

We thus derive

$merge-vars.(" if B' then S1' else S2' fi ", XLs, X, XL1i, XL2f, Y) \triangleq " if B'[XL1i \setminus X1]$

then $merge-vars.(S1', XLs, X, XL1i, XL2f, Y)$ else $merge-vars.(S2', XLs, X, XL1i, XL2f, Y) fi "$ .

Q2. We get $XLs \circ glob.IF$, as required, due to the ind. hypo. (Q2, twice) and since $(XLs \cap glob.B') \subseteq XL1i$ (due to $input$ of IF and P6).
DO

Let $XL_{1i}, XL_{2i}$ be the live-on-entry instances, with $XL_{2i}$ also live-on-exit (which must be same instances as on-entry due to P6 and $\text{ddef.DO'}$ being empty); let $(X_1, X_2) \subseteq X$ be the corresponding program variables (the one-to-one mapping from $XL_{1i}, XL_{2i}$ to $X_1, X_2$ is due to P5). Since all live variables on entry to $\text{DO'}$ are also live on both ends of $S_1'$, we define $S_1 := \text{merge-vars.}(S_1', XLs, X, (XL_{1i}, XL_{2i}), (XL_{1i}, XL_{2i}), Y)$. The validity of P1-P8 of the call to $\text{merge-vars}$ is a consequence of the given P1-P8 (of $\text{DO'}$), of $S_1'$ being a slip of $\text{DO'}$ and of the def. of $\text{ddef}$ and $\text{input}$ of DO loops.

We now aim to use Program equivalence D.4 with the merged $S_1$ and $B := B'[XL_{1i}, XL_{2i} \setminus X_1, X_2]$, and need to show correctness of its P1-P5 preconditions. P1 (disjointness of $(X_1, X_2, XL_{1i}, XL_{2i}, Y)$) is due to P2-P4 and the definition of $XL_{1i}, XL_{2i}$ (only the latter being live-on-exit from $\text{DO'}$); P2 ($(XL_{1i}, XL_{2i}) \circ \text{input.DO}$) is due to P2,P4,Q2 and PR5; P3 is due to our P1 and P6; P4 ($(B' \equiv B'[XL_{1i}, XL_{2i} \setminus X_1, X_2][X_1, X_2 \setminus XL_{1i}, XL_{2i}])$) is due to $(X_1, X_2) \circ \text{glob.B'}$ (our P1,P3,P4) and the redundancy of reversed double sub.; and finally P5 is given by the induction hypothesis (Q1 of $S_1'$); then

$$\text{\"{} \{ (XL_{1i}, XL_{2i}) := (X_1, X_2) ; \text{while } B' \text{ do } S_1' \text{ od } ] [ \text{live } XL_{2i}, Y ] \text{ \"{}} }$$

$$= \text{\{ Program equivalence D.4 with } B := B'[XL_{1i}, XL_{2i} \setminus X_1, X_2] \text{ and } S_1 := \text{merge-vars.(}S_1', XLs, X, (XL_{1i}, XL_{2i}), (XL_{1i}, XL_{2i}), Y)\} \text{\{ while } B \text{ do } S_1 \text{ od } ; XL_{2i} := X_2) [ \text{live } XL_{2i}, Y \text{ \{"} }$$

We thus derive

$$\text{merge-vars.(\"{} \{ while } B' \text{ do } S_1' \text{ od } \text{\"{}} , XLs, X, (XL_{1i}, XL_{2i}), XL_{2i}, Y \} \triangleq \text{\"{}} \text{while } B'[XL_{1i}, XL_{2i} \setminus X_1, X_2] \text{ do merge-vars.(}S_1', XLs, X, (XL_{1i}, XL_{2i}), (XL_{1i}, XL_{2i}), Y) \text{ \"{}} \text{od } \text{\"{}} , \text{\"{}} .$$

Q2. Finally, we get the required $XLs \circ \text{glob.DO}$ due to the ind. hypo. (Q2 on $S_1'$) and since $(XLs \cap \text{glob.B'}) \subseteq (XL_{1i}, XL_{2i})$.

D.4 SSA is de-SSA-able

**Theorem S.4** Let $S$ be any core statement and let $X, Y := (\text{def.S}), (\text{glob.S}\setminus \text{def.S})$, $X_1 := (X \cap ((X \setminus \text{ddef.S}) \cup \text{input.S}))$ and $(XL_{1i}, XL_{f}) := \text{fresh.}((X_1, X), (X, Y))$; let $S'$ be the SSA version of $S$, defined as $S' := \text{toSSA.}(S, X, XL_{1i}, XL_{f}, Y, (XL_{1i}, XL_{f}))$; then $S'$ is de-SSA-able. That is, all
preconditions, P1-P8, of the fromSSA algorithm hold for $S'' := \text{fromSSA}(S', XLs, X, XL1i, XLf, Y)$ where $XLs := ((XL1i, XLf) \cup (\text{def}.S1' \setminus Y))$.

**Proof.** Preconditions P1 ($\text{glob}.S' \subseteq (XLs, Y)$) and P2 ($\left(XL1i \cup XLf\right) \subseteq XLs$) hold by definition of XLs; P3 ($\left(X1 \cup X\right) \subseteq X$) holds by definition of X1 and set theory; P4 $(X \circ (XLs, Y))$ is due to the definition of $X$, $Y$, $XLs$ and due to RE5 (i.e. $\text{def}.S \subseteq \text{glob}.S$) Q2 of toSSA (i.e. $X \circ \text{glob}.S'$); and P6 ($\left(XLs \cap \left((XLf \setminus \text{def}.S') \cup \text{input}.S''\right)\right) \subseteq XL1i$) is due to DP1 of toSSA.

We are left to show no-simultaneous liveness (P5), no-def-on-live (P7) and no-multiple-defs (P8). Those shall be proved by induction over the structure of $S$.

For P5 (no-simultaneous liveness) we first note that having one final instance for each live-on-exit program variable, and similarly having one initial instance for each live-on-entry variable, as is guaranteed by toSSA’s derived property DP1, ensures no-simultaneous liveness on-entry. Thus, in proving P5 for each specific case, we shall only be obliged to show no-simultaneous liveness in internal program points.

**Assignment**

Recall toSSA (" $X4, X2, X5, X6, Y1 := E1, E2, E3, E4, E5$ ", $X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XL4f, XL5f), Y, XLs) \≜ $ " $XL4f, XL2, XL5f, XL6, Y1 := E1', E2', E3', E4', E5'$ " where $(E1', E2', E3', E4', E5', E6') := (E1, E2, E3, E4, E5, E6)$

$[X1, X2, X3, X4 \setminus XL1i, XL2i, XL3i, XL4i]$ and $(XL2, XL6) := \text{fresh}((X2, X6), (X, Y, XLs))$.

P7 (no-def-on-live) and P8 (no-multiple-defs) are both due to the disjointness of $(X2, X4, X5, X6, XL4f, XL5f)$, the freshness of $(XL2, XL4f)$ and hence the disjointness of $(XL2, XL4f, XL5f, XL6)$. Note that $XL4f, XL5f$ are actually live-on-exit, thus not breaking P7.

**Sequential composition**

Recall toSSA (" $S1 \; ; \; S2$ ", $X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XL4f, XL5f), Y, XLs) \≜ $ " $S1' \; ; \; S2'$ " where both $S1', S2'$ are constructed by recursive calls to toSSA.

P5,P7,P8: The induction hypothesis ensures no simultaneous liveness in any point of $S2'$ or $S1'$ and no def-on-live or multiple-defs in any internal assignment slip.
APPENDIX D. SSA

220

IF

toSSA(IF, X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XLAf, XL5f), Y, XLs) \equiv IF'

where IF := “if B then S1 else S2 fi ”,

IF' := “if B[X1, X2, X3, X4 \ XL1i, XL2i, XL3i, XL4i]

then S1' ; XLaf, XL5f := XL4t, XL5t else S2' ; XLaf, XL5f := XL4e, XL5e fi ”,

XL4t := (XL4d1t, XL4d2t, XL4d1d2t),

XL4e := (XL4d1i, XL4d2e, XL4d1d2e),

S1' := toSSA.(S1, X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XL4t, XL5t), Y, XLs')

and S2' := toSSA.(S2, X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XL4e, XL5e), Y, XLs'') .

P5: We have (XL3i, XL4f, XL5f) live at the end of IF' and at the end of both branches. The then branch, ending with assignment “ XLaf, XL5f := XL4t, XL5t ”, yields (XL3i, XL4t, XL5t) at the end of S1' (with one instance for each member of (X3, X4, X5)), and maintains no simultaneous liveness in S1' due to the induction hypothesis. Similarly, the triple (XL3i, XL4e, XL5e) includes on instance for each member of (X3, X4, X5), being live at the end of S2'.

P7,P8: The (pseudo) assignments at the end of both branches of IF' are both to the live-on-exit (XLaf, XL5f), final instances of program variables (X4, X5). This, along with the ind. hypo. on S1' and S2' maintains P7 and P8.

DO

toSSA.(DO, X, (XL1i, XL2i, XL3i, XL4i), (XL3i, XLAf), Y, XLs) \equiv

“ XL2, XLAf := XL2i, XL4i ; DO' ”

where DO := “while B do S1 od ”,

DO' := “while B' do S1' ; XL2, XLAf := XL2b, XL4b od ”

(XL2, XL2b, XL4b) := fresh.((X2, X2, X4), (X, Y, XLs)),

XLs' := (XLs, XL2, XL2b, XL4b),

B' := B[X1, X2, X3, X4 \ XL1i, XL2, XL3i, XL4f]

and S1' := toSSA.(S1, X, (XL1i, XL2, XL3i, XLAf), (XL1i, XL2b, XL3i, XL4b), Y, XLs') .

P5: First, live instances ahead of DO' as well as at the end of its body, are from (XL1i, XL2, XL3i, XLAf), one instance for each member of (X1, X2, X3, X4). This is so due to DP2 of S1' (i.e. input.S1' \ (XL1i, XL2, XL3i, XLAf)) and the definition of B'. Then the assignment at the end of the loop body, “ XL2, XLAf := XL2b, XL4b ”, yields (XL1i, XL2b, XL3i, XL4b) as live instances on exit from S1'. The ind. hypo. ensures no simultaneous liveness in (and ahead of) S1' itself.

P7,P8: As mentioned above, live instances at the end of the loop body are from (XL1i, XL2, XL3i, XLAf). Thus, the assignment there to the live (XL2, XLAf), one instance for
D.5 An SSA-based slice is de-SSA-able

**Theorem 9.6.** Any slide-independent statement from the SSA version of any core statement is de-SSA-able.

That is, let \( S \) be any core statement and let \( X, Y := (\text{def.} S), (\text{glob.} S \setminus \text{def.} S), X_1 := (X \cap ((X \setminus \text{def.} S) \cup \text{input.} S)) \) and \( (XL_1i, XLF) := \text{fresh.}((X1, X), (X, Y)); \) let \( S' \) be the SSA version of \( S \), defined as \( S' := \text{toSSA}.(S, X, XL_1i, XLF, Y, (XL_1i, XLF)); \) let \( XLs := ((XL_1i, XLF) \cup (\text{def.} S1' \setminus Y)) \) be the full set of instances (of \( X \), in \( S' \)) and let \( XLI \) be any (slide-independent) subset of those instances, with final instances \( XL2f := XLI \cap XLF; \) finally let \( SI' := \text{slides.} S'.XLi \) be the corresponding (slide-independent) statement; then \( SI' \) is de-SSA-able. That is, all preconditions, P1-P8, of the fromSSA algorithm hold for \( SI := \text{fromSSA}.(SI', X, XL_1i, XL2f, XLs) \).

**Proof.** Preconditions P1 (\( \text{glob.} S' \subseteq (XLs, Y) \)) and P2 (\( (XL_1i \cup XL2f) \subseteq XLs \)) hold by definition of \( XLs \); P3 ((\( X_1 \cup X_2 \) \( \subseteq X \)) holds by definition of \( X_1 \) (and set theory) and by definition of \( XL2f \) and its mapping to program variables \( X_2 \) in \( X \); and P4 (\( X \circ (XLs, Y) \)) is due to the definition of \( X, Y, XLs \) and due to Q2 of toSSA (\( i.e. \ X \circ \text{glob.} S' \)).

For P5, we observe no-simultaneous liveness is known for \( S'[\text{live} XL2f, Y] \) (Theorem 8.4). This property is preserved by taking the *slides* of any slide-independent set (Corollary C.3). Thus \( (\text{slides.} S'.XLI)[\text{live} XL2f, Y] \) enjoys no-simultaneous liveness for instances of (each member of) \( X \).

For P6 ((\( XLs \cap ((XL2f \setminus \text{def.} S') \cup \text{input.} S') \)) \( \subseteq XL_1i \)), we observe

\[
XLs \cap ((XL2f \setminus \text{def.} (\text{slides.} S'.XLI)) \cup \text{input.}(\text{slides.} S'.XLI))
\]

\[
= \quad \{\text{recall def. of } XLs := ((XL_1i, XLF) \cup (\text{def.} S1' \setminus Y)); \}
\]

\[
\text{let } XLs' := XLs \setminus XL_1i \text{ such that } XLs = (XL_1i, XLs'); \}
\]

\[
\text{note that } XLs' \subseteq \text{def.} S' \text{ since DP2 of toSSA and RE4 give } XLf \subseteq \text{def.} S' \}
\]

\[
(XL_1i, XLs') \cap ((XL2f \setminus \text{def.} (\text{slides.} S'.XLI)) \cup \text{input.}(\text{slides.} S'.XLI))
\]

\[
\subseteq \quad \{\text{set theory} \}
\]

\[
XL_1i \cup (XLs' \cap ((XL2f \setminus \text{def.} (\text{slides.} S'.XLI)) \cup \text{input.}(\text{slides.} S'.XLI))
\]

\[
= \quad \{\text{set theory} \}
\]

\[
XL1i \cup (XLs' \cap ((XL2f \setminus \text{def.} (\text{slides.} S'.XLI)) \cup \text{input.}(\text{slides.} S'.XLI))
\]

\[
\quad \text{we get } \text{input.}(\text{slides.} S'.XLI) \cap \text{def.} S' \subseteq XL1i
\]

\[
XL1i \cup (XL1i \cap \text{def.} S' \cap ((XL2f \setminus \text{def.} (\text{slides.} S'.XLI)) \cup \text{input.}(\text{slides.} S'.XLI)))
\]
\[ \subseteq \{ \text{set theory} \} \]

\[ XL_1 \cup (XL \cap \text{def.} S' \cap ((XL_2f \setminus (XL \cap \text{ddef.}(\text{slides}.S'.XL))) \cup (XL \cap \text{input.}(\text{slides}.S'.XL))) \]

\[ = \{ \text{Lemma C.4 with } S, V, X := S', XL, XL: } \]
\[ \text{indeed } (XL \cap \text{def.} S' \subseteq XL \} \]

\[ XL_1 \cup (XL \cap \text{def.} S' \cap ((XL_2f \setminus (XL \cap \text{ddef.} S'))) \cup (XL \cap \text{input.}(\text{slides}.S'.XL))) \]

\[ = \{ \text{set theory: } XL_2f \subseteq XL \text{ by definition and } XL_2f \subseteq \text{ddef.} S' \text{ by DP2 of } \text{toSSA} \} \]

\[ XL_1 \cup (XL \cap \text{def.} S' \cap (XL \cap \text{input.}(\text{slides}.S'.XL))) \]

\[ \subseteq \{ \text{Lemma C.5 with } S, V, X := S', XL, XL: } \]
\[ \text{indeed } XL \cap \text{def.} S \subseteq XL \} \]

\[ XL_1 \cup (XL \cap \text{def.} S' \cap (XL \cap \text{input.} S')) \]

\[ = \{ XLs \cap \text{input.} S' \subseteq XL_1 \text{ by DP1 of } \text{toSSA}; XL \subseteq XLs \} \]

\[ XL_1 \]

For P7 (no def-on-live), we recall no def-on-live is known for \( S'[\text{live } XL_2f, Y] \) (Theorem 8.4). Like P5, we show that this property is preserved by taking the \text{slides} of any slide-independent set.

Since \( S'[\text{live } XL_2f, Y] \) enjoys the property of no-def-on-live (of XL), and since any assignment slip of the form \( (XL_1 := E1)[\text{live } XL_121, Y] \) in \( (\text{slides}.S'.XL)[\text{live } XL_2f, Y] \) has a corresponding slip \( (XL_1, \text{co} XL_1 := E1, E2)[\text{live } XL_12, Y] \) of \( S'[\text{live } XL_2f, Y] \) with \( XL_121 \subseteq XL_12 \) (due to Theorem C.2), we observe that a defined instance \( x' \in XL_1 \) may only cause a def-on-live violation if another instance \( x'' \) (of the same program variable \( x \)), is live-on-exit from the assignment, \text{i.e.} \( x_2 \in XL_121 \). This can only happen if \( x_2 \in XL_12 \) as well (since \( XL_121 \subseteq XL_12 \)). In such a case, \( x_1 \) already causes a def-on-live violation in the corresponding \( (XL_1, \text{co} XL_1 := E1, E2)[\text{live } XL_12, Y] \), thus contradicting the de-SSA-ability of \( S'[\text{live } XL_2f, Y] \).

Finally, P8 (no-multiple-defs) holds for \( S' \) (see Theorem 8.4) and hence for any of its slides.

\[ \square \]
Appendix E

Final-Use Substitution

E.1 Formal derivation

The following is a formal derivation of final-use substitution, for any core statement \( S \) and matching sets of variables \( X \) and \( X' \).

We begin with “\( S \); \{X = X'\} ” where \( X' \diamond \text{glob}\, S \), and while propagating the assertion backwards into \( S \), as far as possible, we make local assertion-based substitutions to each slip that ends up being preceded by the assertion. Finally, we remove all assertions.

An equality \( x = x' \) will successfully propagate backward over any statement that does not define \( x \); it will also propagate into any IF statement and into those DO loops whose body does not define \( x \).

Side note: in terms of control flow paths, the assertion ends up propagating to (the entry of) any node from which all paths to the exit involve no definition of \( x \). When we are interested in a formulation of cases in which all uses of \( x \) will be substituted, we should be able to express that as follows: all paths to the exit from any use of \( x \) are clear of definitions of \( x \).

\( S = \ " \ X1, Y := E1, E2 \ " \): in deriving \( \langle X1, Y := E1, E2 \rangle[\text{final}-\text{use } X1, X2 \setminus X1', X2'] \) when \( (X1', X2') \diamond \text{glob}\, S \) and \( X2 \diamond Y \), we observe

\[
\begin{align*}
&" \ X1, Y := E1, E2 ; \{X1, X2 = X1', X2'\} " \\
= &\{\text{split assertion (Law 15)}\} \\
&" \ X1, Y := E1, E2 ; \{X2 = X2'\} ; \{X1 = X1'\} " \\
= &\{\text{swap statement and assertion (Law 11)}: (X2, X2') \diamond (X1, Y)\} \\
&" \ {X2 = X2'} ; \ X1, Y := E1, E2 ; \{X1 = X1'\} " \\
= &\{\text{assertion-based sub. (Law 17)}: E1' := E1[X2 \setminus X2'] \text{ and } E2' := E2[X2 \setminus X2']\} \\
&" \ {X2 = X2'} ; \ X1, Y := E1', E2' ; \{X1 = X1'\} "
\end{align*}
\]
APPENDIX E. FINAL-USE SUBSTITUTION

\[ (\text{final-use sub.}) \]

\[ (X, Y := E_1, E_2 \mid X_1 = X_2', \{X_1 = X_1'\}) \]

We thus derive \((X, Y := E_1, E_2) \mid \text{final-use } X_1, X_2 \setminus X_1', X_2'\) \(\triangleq\)

\[ (X, Y := E_1, E_2 \mid X_1 = X_2', \{X_1 = X_1'\}) \]

\(S = \{S_1; S_2\}\): in deriving \((S_1; S_2) \mid \text{final-use } X_1, X_2 \setminus X_1', X_2'\) when \(X_2 \circ \text{def.} S_2\) and \((X_1', X_2') \circ \text{glob.} S\), we observe

\[ (S_1; S_2 \mid \{X_1, X_2 = X_1', X_2'\}) \]

\[ (\text{split assertion (Law 15)}) \]

\[ (S_1; S_2' \mid \{X_2 = X_2'\} \setminus \{X_1 = X_1'\}) \]

\[ (\text{swap statement and assertion (Law 11): } (X, X_2') \circ \text{def.} S_2) \]

\[ (S_1; S_2' \mid \{X_2 = X_2'\} \setminus \{X_1 = X_1'\}) \]

\[ (\text{merge assertions (Law 15)} \]

\[ (S_1' \setminus \{X_2 = X_2'\} \setminus \{X_1 = X_1'\}) \]

\[ (S_1'; S_2' \mid \{X_2 = X_2'\} \setminus \{X_1 = X_1'\}) \]

\[ (S_1'; S_2' \mid \{X_1, X_2 = X_1', X_2'\}) \]

We thus derive \((S_1; S_2) \mid \text{final-use } X_1, X_2 \setminus X_1', X_2'\) \(\triangleq\)

\[ (S_1' \setminus \{X_2 = X_2'\} \setminus \{X_1 = X_1'\}) \]

\(S = \{\text{dist. assertion over IF (Law 13)}\]

\[ (\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}) \mid \text{final-use } X_1, X_2 \setminus X_1', X_2'\]

when \(X_2 \circ (\text{def.} S_1 \cup \text{def.} S_2)\) and \((X_1', X_2') \circ \text{glob.} S\), we observe

\[ (\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}) \]

\[ (\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}) \]

\[ (\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}) \]

\[ (\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}) \]

\[ (\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}) \]

\[ (\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}) \]

\[ (\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}) \]
\[ \{ \text{final-use sub., twice: let } S1' := S1[\text{final-use } X1, X2 \setminus X1', X2'] \text{ and } S2' := S2[\text{final-use } X1, X2 \setminus X1', X2'] \} \]

if \( B \) then \( S1' \); \( \{ X1, X2 = X1', X2' \} \) else \( S2' \); \( \{ X1, X2 = X1', X2' \} \) fi \)

= \{ \text{dist. IF over ' ; ' (Law 4)} \}

if \( B \) then \( S1' \) else \( S2' \) fi ; \( \{ X1, X2 = X1', X2' \} \)

= \{ \text{split assertion (Law 15)} \}

if \( B \) then \( S1' \) else \( S2' \) fi ; \( \{ X2 = X2' \} \); \( \{ X1 = X1' \} \)

= \{ \text{swap statement and assertion (Law 11): } (X2, X2') \circ (\text{def.} S1' \cup \text{def.} S2') \}

\{ \{ X2 = X2' \} ; \text{if } B \text{ then } S1' \text{ else } S2' \text{ fi} ; \{ X1 = X1' \} \}

= \{ \text{assertion-based sub. (Law 17)} \}

\{ \{ X2 = X2' \} ; \text{if } B[X2 \setminus X2'] \text{ then } S1' \text{ else } S2' \text{ fi} ; \{ X1 = X1' \} \}

= \{ \text{swap assertion and statement (Law 11): } (X2, X2') \circ (\text{def.} S1' \cup \text{def.} S2') \}

\{ \{ X2 = X2' \} ; \text{if } B[X2 \setminus X2'] \text{ then } S1' \text{ else } S2' \text{ fi} ; \{ X1 = X1' \} \}

= \{ \text{merge assertions (Law 15)} \}

\{ \{ X2 = X2' \} ; \text{if } B[X2 \setminus X2'] \text{ then } S1' \text{ else } S2' \text{ fi} ; \{ X1, X2 = X1', X2' \} \} .

We thus derive \((B \text{ then } S1 \text{ else } S2 \text{ fi})[\text{final-use } X1, X2 \setminus X1', X2'] \triangleq \)

\( \{ \text{if } B[X2 \setminus X2'] \text{ then } S1[\text{final-use } X1, X2 \setminus X1', X2'] \text{ else } S2[\text{final-use } X1, X2 \setminus X1', X2'] \text{ fi} \} \) where \( X1 := X \cap \text{def.}(S1, S2) \), \( X2 := X \setminus X1 \) and \( X1', X2' \) are the corresponding subsets of \( X' \).

S = “\( \text{while } B \text{ do } S1 \text{ od} \)” in deriving \((\text{while } B \text{ do } S1 \text{ od})[\text{final-use } X1, X2 \setminus X1', X2']\)

(when \( X2 \circ \text{def.} S1 \) and \( (X1', X2') \circ \text{glob.} S \)), we observe

\( \{ \text{while } B \text{ do } S1 \text{ od} ; \{ X1, X2 = X1', X2' \} \} \)

= \{ \text{split assertion (Law 15)} \}

\( \{ \text{while } B \text{ do } S1 \text{ od} ; \{ X2 = X2' \} ; \{ X1 = X1' \} \} \)

= \{ \text{swap statement and assertion (Law 11): } (X2, X2') \circ \text{def.} S1 \}

\{ \{ X2 = X2' \} ; \text{while } B \text{ do } S1 \text{ od} ; \{ X1 = X1' \} \}

= \{ \text{prop. assertion forward into loop (Law 16): } (X2, X2') \circ \text{def.} S1 \}

\{ \{ X2 = X2' \} ; \text{while } B \text{ do } \{ X2 = X2' \} ; S1 \text{ od} ; \{ X1 = X1' \} \}

= \{ \text{swap assertion and statement (Law 11): } (X2, X2') \circ \text{def.} S1 \}

\{ \{ X2 = X2' \} ; \text{while } B \text{ do } S1 ; \{ X2 = X2' \} \text{ od} ; \{ X1 = X1' \} \}

= \{ \text{final-use sub.: let } S1' := S1[\text{final-use } X2 \setminus X2'] \}
“\{X2 = X2’\} ; while B do S1 ; \{X2 = X2’\} od ; \{X1 = X1’\} ”

= \{assertion-based sub. (Law 17): let B’ := B[X2 \ X2’]\}

“\{X2 = X2’\} ; while B’ do S1 ; \{X2 = X2’\} od ; \{X1 = X1’\} ”

= \{swap statement and assertion (Law 11): (X2, X2’) \circ def.S1’\}

“\{X2 = X2’\} ; while B’ do \{X2 = X2’\} ; S1’ od ; \{X1 = X1’\} ”

= \{prop. assertion backward outside loop (Law 16): (X2, X2’) \circ def.S1’\}

“\{X2 = X2’\} ; while B’ do S1’ od ; \{X1 = X1’\} ”

= \{merge assertions (Law 15)\}

“\{X2 = X2’\} ; while B’ do S1’ od ; \{X1, X2 = X1’, X2’\} ” .

We thus derive \((\text{while } B \text{ do } S1 \text{ od})[\text{final-use } X1, X2 \setminus X1’, X2’] \triangleq (\text{while } B[X2 \setminus X2’] \text{ do } S1[\text{final-use } X2 \setminus X2’] \text{ od})\) where \(X1 := X \cap def.S1, X2 := X \setminus X1\) and \(X2’\) is the subset of \(X’\) corresponding to \(X2\).

E.2 Lemmata for proving statement dup. with final use

**Lemma 10.2.** Let \(S\) be any core statement with \(def.S = (V, coV), Vr \subseteq V\) (and \(fVr\) the corresponding subset of \(fV\)) and \((iV, icoV, fV) \circ glob.S\); we then have

\[
\begin{array}{|c|c|}
\hline
\text{“ } iV, icoV := V, coV & V, coV := iV, icoV \\
\hline
\text{; } S & \text{; } S[\text{final-use } Vr \setminus fVr] \\
\hline
\text{; } fV := V & \text{; } \{Vr = fVr\} ” \\
\hline
\end{array}
\]

= 

\[
\begin{array}{|c|c|}
\hline
\text{“ } iV, icoV := V, coV & V, coV := iV, icoV \\
\hline
\text{; } S & \text{; } S[\text{final-use } Vr \setminus fVr] \\
\hline
\text{; } fV := V & ” \\
\hline
\end{array}
\]
APPENDIX E. FINAL-USE SUBSTITUTION

Proof.

\[
\begin{align*}
&\qquad \text{"} iV, icoV := V, coV ; S ; fV := V ; \\
&\quad V, coV := iV, icoV ; S[\text{final-use } Vr \setminus fVr] ; \{ Vr = fVr \} \\
&= \quad \{ \text{swap statement and assertion (Law 10)} \}
\end{align*}
\]

\[
\begin{align*}
&\qquad \text{"} iV, icoV := V, coV ; S ; fV := V ; \\
&\quad V, coV := iV, icoV ; \{ wp.S[\text{final-use } Vr \setminus fVr].(Vr = fVr) \} ; \\
&\quad S[\text{final-use } Vr \setminus fVr] \\
&= \quad \{ \text{property of final-use sub. (Lemma E.1 see below)} \}
\end{align*}
\]

\[
\begin{align*}
&\qquad \text{"} iV, icoV := V, coV ; S ; fV := V ; \\
&\quad V, coV := iV, icoV ; \{ wp.S. (Vr = fVr) \} ; S[\text{final-use } Vr \setminus fVr] \\
&= \quad \{ \text{intro. following assertion (Law 7)} \}
\end{align*}
\]

\[
\begin{align*}
&\qquad \text{"} iV, icoV := V, coV ; \\
&\quad \{ wp." \{ iV, icoV = V, coV \} ; S ; fV := V ; \\
&\quad \quad V, coV := iV, icoV ; S[\text{final-use } Vr \setminus fVr] \}
\end{align*}
\]

\[
\begin{align*}
&\qquad \{ \text{swap statement and assertion (Law 10) and wp of } \}' \}
\end{align*}
\]

\[
\begin{align*}
&\qquad \text{"} iV, icoV := V, coV ; \\
&\quad \{ \text{wp." \{ iV, icoV = V, coV \} ; S " . true } \} ; \\
&\quad \quad (\{ iV, icoV = V, coV \} ; S ) ; fV := V ; \\
&\quad \quad \quad V, coV := iV, icoV ; S[\text{final-use } Vr \setminus fVr] \\
&= \quad \{ \text{Lemma E.2 see below} \}
\end{align*}
\]

\[
\begin{align*}
&\qquad \text{"} iV, icoV := V, coV ; \{ wp." \{ iV, icoV = V, coV \} ; S " . \text{true } \} ; \\
&\quad \quad (\{ iV, icoV = V, coV \} ; S ) ; fV := V ; \\
&\quad \quad \quad V, coV := iV, icoV ; S[\text{final-use } Vr \setminus fVr] \\
&= \quad \{ \text{swap assertion and statement (Law 10)} \}
\end{align*}
\]

\[
\begin{align*}
&\qquad \text{"} iV, icoV := V, coV ; \{ (iV, icoV = V, coV ) ; S ) ; fV := V ; \\
&\quad \quad V, coV := iV, icoV ; S[\text{final-use } Vr \setminus fVr] \\
&= \quad \{ \text{remove true assertion; remove following assertion (Law 7)} \}
\end{align*}
\]

\[
\begin{align*}
&\qquad \text{"} iV, icoV := V, coV ; S ; fV := V ; \\
&\quad \quad V, coV := iV, icoV ; S[\text{final-use } Vr \setminus fVr] \\
&= \quad \text{\{ remove true assertion; remove following assertion (Law 7) \}}
\end{align*}
\]

\[
\begin{align*}
\Box
\end{align*}
\]

Lemma E.1. Let $S, V, fV$ be any core statement and two sets of variables, respectively; we then have

\[
[wp.(S[\text{final-use } V \setminus fV].(V = fV) \equiv wp.S.(V = fV)]
\]
provided \( fV \circ (V \cup \text{glob}.S) \).

**Proof.**

\[
\begin{align*}
\text{wp}.S.(V = fV) &= \{\text{pred. calc.}\} \\
\text{wp}.S.((V = fV) \land \text{true}) &= \{\text{wp of assertions}\} \\
\text{wp}.S.(\text{wp}^{\text{true}} \{V = fV\} ) &= \{\text{wp of ';'}\} \\
\text{wp}^{\text{true}} S ; \{V = fV\} &= \{\text{final-use sub.}\} \\
\text{wp}^{\text{true}} S[\text{final-use } V \setminus fV] ; \{V = fV\} &= \{\text{wp of assertions}\} \\
\text{wp}^{\text{true}} S[\text{final-use } V \setminus fV].((V = fV) \land \text{true}) &= \{\text{pred. calc.}\} \\
\text{wp}^{\text{true}} S[\text{final-use } V \setminus fV].(V = fV).
\end{align*}
\]

\[\blacksquare\]

**Lemma E.2.** Let \( S \) be any core statement with \( \text{def}.S = (V, coV) \). We then have

\[
\begin{align*}
\text{wp}^{\text{true}} \{iV, icoV = V, coV\} ; S ; fV &:= V ; V, coV := iV, icoV ; S \quad .(Vr = fVr) \\
\text{wp}^{\text{true}} \{iV, icoV = V, coV\} ; S ; fV &:= V \quad .(Vr = fVr) \\
\text{wp}^{\text{true}} \{iV, icoV = V, coV\} ; fV &:= V \quad .(Vr = fVr) \\
\text{wp}^{\text{true}} \{iV, icoV = V, coV\} ; S &\quad .((fV := V).(Vr = fVr)) \\
\text{wp}^{\text{true}} \{iV, icoV = V, coV\} ; S &\quad .\text{true}
\end{align*}
\]

provided \( Vr \subseteq V \), \( fVr \) is the corresponding subset of \( fV \) and \((iV, icoV, fV) \circ \text{glob}.S\).
E.3 Stepwise final-use substitution

Theorem E.3. The final-use substitution can be performed in a stepwise manner. That is, for any core statement $S$ and four sets of variables $X_1, X_2, fX_1, fX_2$, we have

$$S[\text{final-use } X_1, X_2 \setminus fX_1, fX_2] = S[\text{final-use } X_1 \setminus fX_1][\text{final-use } X_2 \setminus fX_2]$$ provided $(fX_1, fX_2) \circ \text{glob}. S$.

Proof. We follow the semantic requirement of final-use substitution ($“S ; \{X = fX\}” = “S[\text{final-use } X \setminus fX] ; \{X = fX\}”$) and observe

$“S[\text{final-use } X_1, X_2 \setminus fX_1, fX_2] ; \{X_1, X_2 = fX_1, fX_2\}”$

$= \{\text{final-use sub.: } (fX_1, fX_2) \circ \text{glob}. S (proviso)\}$

$“S ; \{X_1, X_2 = fX_1, fX_2\}”$

$= \{\text{split assertion: Law 15}\}$

$“S ; \{X_1 = fX_1\} ; \{X_2 = fX_2\}”$

$= \{\text{final-use sub.: } fX_1 \circ \text{glob}. S (proviso)\}$

$“S[\text{final-use } X_1 \setminus fX_1] ; \{X_1 = fX_1\} ; \{X_2 = fX_2\}”$

$= \{\text{swap statements (Program equivalence 5.7): def of assertions is empty}\}$

$“S[\text{final-use } X_1 \setminus fX_1] ; \{X_2 = fX_2\} ; \{X_1 = fX_1\}”$

$= \{\text{final-use sub.: } fX_2 \circ \text{glob}. S[\text{final-use } X_1 \setminus fX_1] \text{ since } fX_2 \circ (fX_1, \text{glob}. S) \text{ (as implied by the proviso)}\}$

$“S[\text{final-use } X_1 \setminus fX_1][\text{final-use } X_2 \setminus fX_2] ; \{X_2 = fX_2\} ; \{X_1 = fX_1\}”$

$= \{\text{merge assertions: Law 15}\}$

$“S[\text{final-use } X_1 \setminus fX_1][\text{final-use } X_2 \setminus fX_2] ; \{X_1, X_2 = fX_1, fX_2\}”$.

$\square$
Appendix F

Summary of Laws

F.1 Manipulating core statements

**Law 1.** Let $X, Y, E_1, E_2$ be two sets of variables and two sets of expressions, respectively; then

\[
\{X := E_1 ; Y := E_2\} = \{X, Y := E_1, E_2\}
\]

provided $X \diamond ((Y \cup \text{glob}.E_2)$.

**Law 2.** Let $S, X$ be a statement set of variables, respectively; then

\[
S = \{S; X := X\}
\]

**Law 3.** Let $S, S_1, S_2, B$ be three statements and a boolean expression, respectively; then

\[
\{S ; \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi }\} = \{\text{if } B \text{ then } S ; S_1 \text{ else } S ; S_2 \text{ fi}\}
\]

provided $\text{def}.S \circ \text{glob}.B$.

**Law 4.** Let $S_1, S_2, S_3, B$ be three statements and a boolean expression, respectively; then

\[
\{\text{if } B_1 \text{ then } S_1 \text{ else } S_2 \text{ fi ; } S_3\} = \{\text{if } B_1 \text{ then } S_1 \text{ ; } S_3 \text{ else } S_2 \text{ ; } S_3 \text{ fi}\}
\]

**Law 5.** Let $S_1, X, B, E$ be any statement, set of variables, boolean expression and set of expressions, respectively; then

\[
\{X := E\} ; \text{while } B \text{ do } S_1 ; (X := E) \text{ od}\} = \{X := E\} ; \text{while } B \text{ do } S_1 \text{ od} ; (X := E)
\]

provided $X \circ (\text{glob}.B \cup \text{input}.S_1 \cup \text{glob}.E)$.

**Law 6.** Let $X, E$ be any set of variables and set of expressions, respectively; then

\[
\{X := E\} = \{X := E\} ; X := E
\]
F.2 Assertion-based program analysis

F.2.1 Introduction of assertions

Law 7. Let \( X, Y, E_1, E_2 \) be two sets of variables and two sets of expressions, respectively; then

\[
\text{"} X, Y := E_1, E_2 \; \text{"} = \text{"} X, Y := E_1, E_2 ; \{ Y = E_2 \} \; \text{"}
\]

provided \((X, Y) \circ \text{glob.} E_2\).

Law 8. Let \( X, X', E \) be (same length) lists of variables and expressions, respectively, with \( X \circ X' \); then

\[
\text{"} X, X' := E, E \; \text{"} = \text{"} X, X' := E, E ; \{ X = X' \} \; \text{"}.
\]

Law 9. Let \( S_1, B_1, B_2 \) be any given statement and two boolean expressions, respectively; then

\[
\text{"} \text{while} \ B_1 \ \text{do} \ S_1 \ \text{od} \; \text{"} = \text{"} \text{while} \ B_1 \ \text{do} \ \{ B_2 \} ; S_1 \ \text{od} \; \text{"}.
\]

provided \([B_1 \Rightarrow B_2]\).

F.2.2 Propagation of assertions

Law 10. Let \( S, B \) be a statement and boolean expression, respectively; then

\[
\text{"} \{ \text{wp.} S. B \} ; S \; \text{"} = \text{"} S ; \{ B \} \; \text{"}.
\]

Law 11. Let \( S, B \) be a statement and boolean expression, respectively; then

\[
\text{"} \{ B \} ; S \; \text{"} = \text{"} S ; \{ B \} \; \text{"}.
\]

provided \(\text{def.} S \circ \text{glob.} B\).

Law 12. Let \( S_1, S_2, B_1, B_2 \) be two statements and two boolean expressions, respectively; then

\[
\text{"} \{ B_1 \} ; \text{if} \ B_2 \ \text{then} \ S_1 \ \text{else} \ S_2 \ \text{fi} \; \text{"} = \text{"} \text{if} \ B_2 \ \text{then} \ \{ B_1 \} ; S_1 \ \text{else} \ \{ B_1 \} ; S_2 \ \text{fi} \; \text{"}.
\]

Law 13. Let \( S, B_1, B_2, B_3, B_4 \) be a statement and four boolean expressions, respectively; then

\[
\text{"} \{ B_1 \} ; \text{while} \ B_2 \ \text{do} \ S ; \{ B_3 \} \ \text{od} \; \text{"} = \text{"} \{ B_1 \} ; \text{while} \ B_2 \ \text{do} \ \{ B_4 \} ; S ; \{ B_3 \} \ \text{od} \; \text{"}
\]

provided \([B_1 \Rightarrow B_4]\) and \([B_3 \Rightarrow B_4]\).

Law 14. Let \( S, B_1, B_2, B_3, B_4 \) be a statement and four boolean expressions, respectively; then

\[
\text{"} \{ B_1 \} ; \text{while} \ B_2 \ \text{do} \ S ; \{ B_3 \} \ \text{od} \; \text{"} = \text{"} \{ B_1 \} ; \text{while} \ B_2 \land B_4 \ \text{do} \ S ; \{ B_3 \} \ \text{od} \; \text{"}
\]

provided \([B_1 \Rightarrow B_4]\) and \([B_3 \Rightarrow B_4]\).
Law 15. Let $B_1, B_2$ be two boolean expressions; then

\[ \{ B_1 \land B_2 \} = \{ B_1 \} \cup \{ B_2 \} \]

Law 16. Let $S, B_1, B_2$ be a statement and two boolean expressions, respectively; then

\[ \{ B_1 \} \land \text{while } B_2 \text{ do } S \text{ od } = \{ B_1 \} \land \text{while } B_2 \text{ do } \{ B_1 \} \land S \text{ od } \]

provided \( \text{glob}.B_1 \circ \text{def}.S \).

F.2.3 Substitution

Law 17. Let $S_1, S_2, B$ be two statements and a boolean expression, respectively; let $X, E$ be a set of variables and a corresponding list of expressions; and let $Y, Y'$ be two sets of variables; then

\[ \{ Y = Y' \} ; X := E \} = \{ Y = Y' \} ; X := E[Y \setminus Y'] \}

\[ \{ Y = Y' \} ; \text{IF} \} = \{ Y = Y' \} ; \text{IF}' \}

\[ \{ Y = Y' \} ; \text{DO} \} = \{ Y = Y' \} ; \text{DO}' \}

where \( \text{IF} \) := “ if $B$ then $S_1$ else $S_2$ fi ”,

\[ \text{IF}' := \text{if } B[Y \setminus Y'] \text{ then } S_1 \text{ else } S_2 \text{ fi } ”, \]

\[ \text{DO} := \text{while } B \text{ do } S_1 \; \{ Y = Y' \} \text{ od } ”, \]

and \( \text{DO}' := \text{while } B[Y \setminus Y'] \text{ do } S_1 \; \{ Y = Y' \} \text{ od } ”. \]

Law 18. Let $S_1, S_2, B$ be two statements and a boolean expression, respectively; let $X, X', Y, Z, E_1, E_1', E_2, E_3$ be four lists of variables and corresponding lists of expressions; then

\[ \{ X, Y := E_1, E_2 \; ; \; Z := E_3 \} = \{ X, Y := E_1, E_2 \; ; \; Z := E_3[Y \setminus E_2] \}

\[ \{ X, Y := E_1, E_2 \; ; \; \text{IF} \} = \{ X, Y := E_1, E_2 \; ; \; \text{IF}' \}

\[ \{ X, Y := E_1, E_2 \; ; \; \text{DO} \} = \{ X, Y := E_1, E_2 \; ; \; \text{DO}' \}

provided \((X \cup X'), Y) \circ \text{glob}.E_2 \)

where \( \text{IF} := \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } ”,

\[ \text{IF}' := \text{if } B[Y \setminus E_2] \text{ then } S_1 \text{ else } S_2 \text{ fi } ”, \]

\[ \text{DO} := \text{while } B \text{ do } S_1 \; X', Y := E_1', E_2 \text{ od } ”, \]

and \( \text{DO}' := \text{while } B[Y \setminus E_2] \text{ do } S_1 \; X', Y := E_1', E_2 \text{ od } ”. \)

F.3 Live variables analysis

F.3.1 Introduction and removal of liveness information

Law 19. Let $S, V$ be any statement and set of variables, respectively, with \( \text{def}.S \subseteq V \); then

\[ S = \{ S[\text{live } V] \} \]
F.3.2 Propagation of liveness information

Law 20. Let $S_1, S_2, V_1, V_2$ be any two statements and two sets of variables, respectively; then

$$\text{“} (S_1 ; S_2)[live V_1] \text{”} = \text{“} (S_1[live V_2] ; S_2[live V_1])[live V_1] \text{”}$$

provided $V_2 = (V_1 \setminus ddef.S_2) \cup input.S_2$.

Law 21. Let $B, S_1, S_2, V$ be any boolean expression, two statements and set of variables, respectively; then

$$\text{“} (if B then S_1 else S_2 fi)[live V] \text{”} = \text{“} (if B then S_1[live V] else S_2[live V] fi)[live V] \text{”}$$

Law 22. Let $B, S, V$ be any boolean expression, statement and set of variables, respectively; then

$$\text{“} (while B do S od)[live V_1] \text{”} = \text{“} (while B do S[live V_2] od)[live V_1] \text{”}$$

provided $V_2 = V_1 \cup (glob.B \cup input.S)$.

F.3.3 Dead assignments: introduction and elimination

Law 23. Let $S, V, X, Y, E_1, E_2$ be any statement, three sets of variables and two sets of expressions, respectively; then

$$\text{“} (S ; X := E_1)[live V] \text{”} = \text{“} (S ; X, Y := E_1, E_2)[live V] \text{”}$$

provided $Y \diamond (X \cup V)$.

Law 24. Let $S, V, Y, E$ be any statement, two sets of variables and set of expressions, respectively; then

$$\text{“} S[live V] \text{”} = \text{“} (S ; Y := E)[live V] \text{”}$$

provided $Y \diamond V$.

Law 25. Let $S, V, X, Y, E_1, E_2$ be any statement, three sets of variables and two sets of expressions, respectively; then

$$\text{“} (X := E_1 ; S)[live V] \text{”} = \text{“} (X, Y := E_1, E_2 ; S)[live V] \text{”}$$

provided $Y \diamond (X \cup (V \setminus ddef.S) \cup input.S)$.

Law 26. Let $B, S_1, S_2, Y, V, E$ be a boolean expression, two statements, two sets of variables and a set of expressions, respectively; then

$$\text{“} (S_1 ; while B do S_2 od)[live V] \text{”} = \text{“} (S_1 ; while B do S_2 ; (Y := E) od)[live V] \text{”}$$

provided $Y \diamond (V \cup glob.B \cup input.S_2)$. 
Bibliography


[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.


[72] Yahoo Refactoring Group Mailing List. refactoring@yahoogroups.com. 12