ABSTRACT

Dynamic software update (DSU) enables a program to update while it is running. DSU aims to minimize the loss due to program downtime for updates. Usually DSU is done in three steps: suspending the execution of an old program, mapping the execution state from the old program to a new one, and resuming execution of the new program with the mapped state. The semantic correctness of DSU depends largely on the state mapping which is mostly composed by developers manually nowadays. However, the manual construction of a state mapping does not necessarily ensure sound and dependable state mapping. This dissertation presents a methodology to assist developers by automating the construction of a partial state mapping with a guarantee of correctness.

This dissertation includes a detailed study of DSU correctness and automatic state mapping for server programs with an established user base. At first, the dissertation presents the formal treatment of DSU correctness and the state mapping problem. Then the dissertation presents an argument that for programs with an established user base, dynamic updates must be backward compatible. The dissertation next presents a general definition of backward compatibility that specifies the allowed changes in program interaction between an old version and a new version and identified patterns of code evolution that results in backward compatible behavior. Thereafter the dissertation presents formal definitions of these patterns together with proof that any changes to programs in these patterns will result in backward compatible update. To show the applicability of the results, the dissertation presents SitBack, a program analysis tool that has an old version program and a new one as input and computes a partial state mapping under the assumption that the new version is backward compatible with the old version. SitBack does not handle all kinds of changes and it reports to the user in incomplete part of a state mapping. The dissertation presents a
detailed evaluation of SitBack which shows that the methodology of automatic state mapping is promising in dealing with real-world program updates. For example, SitBack produces state mappings for 17–75% of the changed functions. Furthermore, SitBack generates automatic state mapping that leads to successful DSU. In conclusion, the study presented in this dissertation does assist developers in developing state mappings for DSU by automating the construction of state mappings with a correctness guarantee, which helps the adoption of DSU ultimately.
To my family
ACKNOWLEDGMENTS

I am grateful to my advisor, other members of my dissertation committee, my TA work supervisor and, most importantly, my family for the guidance, help and support of my PhD study in Arizona State University.

First, I am deeply gratitude to my advisor Dr. Rida A. Bazzi for his patient guidance and encouragement for my PhD study. I learnt from Dr. Bazzi how to think critically and effectively communicate in academic research. I also learnt the importance of attention to details from Dr. Bazzi. I remember the meetings with Dr. Bazzi for him to correct my awful writing. Without Dr. Bazzi’s careful and patient mentorship, I could not finish this dissertation. I owe a lot to Dr. Bazzi.

Second, I would like to thank my dissertation committee members Dr. Georgios Fainekos, Dr. Iulian Neamtiu and Dr. Aviral Shrivastava for their insightful advice and suggestions to my Ph.D study. I appreciate their time and effort in serving on my dissertation committee. In addition, I am grateful to Dr. Violet R. Syrotiuk for the opportunity to work with Dr. Syrotiuk as a teaching assistant for a long time. I am also grateful to Dr. Mutsumi Nakamura and Pamela Dunn for the help with my application for a TA position and multiple extensions of my I-20.

Third, I would like to thank my friends during my stay in ASU: Dejun Yang, Ziming Zhao, Xi Fang, Jin Zhang, Wu Li, Yichuan Cai, Peide Zhong, Satyajayant Misra, Yunzhong Liu. You guys made my PhD study enjoyable.

Last but not least, I want to express my gratitude to my parents. During my long PhD study, my parents showed their unwavering support, encouragement and love to me. I am indebted a lot to my parents for their relentless love. I am fortunate to meet Tingyi who bring a lot of needed sunshine in my life. I owe a lot to her and my parents in law for the improvement of my communication skill. Thanks Tingyi for your encouragement when I was stressful.
This dissertation is based upon work supported in part by the National Science Foundation under Grant No. Grant CSR-0849980. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
</tbody>
</table>

CHAPTER

1 INTRODUCTION .................................................. 1
   1.1 Contributions ........................................... 5
   1.2 Organization ............................................ 6

2 RELATED WORK .................................................. 7
   2.1 General Dynamic Software Update System .................. 7
   2.2 DSU Correctness Criteria ................................ 13
   2.3 Automatic Generation of State Mapping ................... 15

3 DYNAMIC SOFTWARE UPDATE AND BACKWARD COMPATIBILITY ...... 17
   3.1 Program, Execution and Specification ..................... 17
   3.2 Hybrid Execution, State Mapping and Backward Compatibility 20
   3.3 Backward Compatible Program Behavior Changes ........... 25

4 A FRAMEWORK OF SEMANTIC EQUIVALENCE FOR INTERACTIVE PROGRAMS ........................................ 29
   4.1 Formal Programming Language ................................ 29
      4.1.1 Language Syntax ..................................... 29
      4.1.2 Small-Step Operational Semantics ..................... 31
   4.2 Execution .................................................. 38
      4.2.1 Terminating Execution and Non-terminating Execution 39
   4.3 Equivalent Computation for Terminating Programs ........ 39

vi
CHAPTER 4.3.1 Proof Rule for Equivalent Computation for Terminating
Programs ......................................................... 40
4.3.2 Soundness of the Proof Rule for Equivalent Computation
for Terminating Programs ................................. 43
4.3.3 Supporting Lemmas for the Soundness Proof of Equivalent
Computation for Terminating Programs ................. 44
4.4 Termination in the Same Way ............................. 46
4.4.1 Proof Rule for Termination in the Same Way ............ 47
4.4.2 Soundness of the Proof Rule for Termination in the Same
Way ............................................................... 51
4.4.3 Supporting Lemmas for the Soundness Proof of Termination
in the Same Way ................................................ 52
4.5 Behavioral Equivalence ....................................... 63
4.5.1 Proof Rule for Behavioral Equivalence .................... 63
4.5.2 Soundness of the Proof Rule for Behavioral Equivalence ... 67
4.5.3 Supporting Lemmas for the Soundness Proof of Behavioral
Equivalence ..................................................... 67
4.6 Backward Compatibility of Equivalent Programs .............. 74
5 REAL WORLD BACKWARD COMPATIBLE PROGRAM UPDATES . 76
5.1 Summary of a Study of Real World Program Evolution ........ 76
5.1.1 Observational Equivalence: the Old Behavior ............ 77
5.1.2 Enumeration Type Extension: Old Behavior for Old Input
and Allowing New Input ........................................ 81
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.3 Variable Type Weakening: More Output When the Old Program Terminates</td>
<td>81</td>
</tr>
<tr>
<td>5.1.4 Exit on Errors: Stopping Execution While the Old Program Produces More Output</td>
<td>82</td>
</tr>
<tr>
<td>5.1.5 Improved Prompt Messages: Functionally Equivalent Outputs</td>
<td>83</td>
</tr>
<tr>
<td>5.1.6 Missing Variable Initialization: Enforcing Restrictions on Program States</td>
<td>83</td>
</tr>
<tr>
<td>5.2 Proof Rule for Specializing New Configuration Variables</td>
<td>84</td>
</tr>
<tr>
<td>5.3 Proof Rule for Enumeration Type Extension</td>
<td>88</td>
</tr>
<tr>
<td>5.4 Proof Rule for Variable Type Weakening</td>
<td>93</td>
</tr>
<tr>
<td>5.5 Proof Rule for Exit on Errors</td>
<td>95</td>
</tr>
<tr>
<td>5.6 Proof Rule for Improved Prompt Message</td>
<td>100</td>
</tr>
<tr>
<td>5.7 Proof Rule for Missing Variable Initializations</td>
<td>105</td>
</tr>
<tr>
<td>6 SITBACK: THE STATE MAPPING TOOL</td>
<td>107</td>
</tr>
<tr>
<td>6.1 Motivating Example</td>
<td>107</td>
</tr>
<tr>
<td>6.2 Automatic Calculation of State Mapping for Real World Programs</td>
<td>111</td>
</tr>
<tr>
<td>6.3 Methodology of Generating State Mapping</td>
<td>113</td>
</tr>
<tr>
<td>6.3.1 Matching Individual Functions</td>
<td>114</td>
</tr>
<tr>
<td>6.3.2 Matching Whole Programs</td>
<td>117</td>
</tr>
<tr>
<td>6.4 Implementation</td>
<td>118</td>
</tr>
<tr>
<td>6.4.1 Alias Analysis</td>
<td>119</td>
</tr>
<tr>
<td>6.4.2 Semantic Analyzer</td>
<td>122</td>
</tr>
<tr>
<td>6.4.3 PDG/Slice Generation</td>
<td>123</td>
</tr>
<tr>
<td>6.4.4 PDG Matching</td>
<td>125</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>6.4.5</td>
<td>Patch Generation</td>
</tr>
<tr>
<td>6.5</td>
<td>Limitations</td>
</tr>
<tr>
<td>6.6</td>
<td>Evaluation</td>
</tr>
<tr>
<td></td>
<td>6.6.1 Versatility Evaluation</td>
</tr>
<tr>
<td></td>
<td>6.6.2 Applicability Evaluation</td>
</tr>
<tr>
<td>7</td>
<td>CONCLUSION</td>
</tr>
<tr>
<td></td>
<td>7.1 Future Work</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
</tr>
<tr>
<td>APPENDIX</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>TYPE SYSTEM OF THE PROGRAMMING LANGUAGE</td>
</tr>
<tr>
<td>B</td>
<td>SUPPORTING LEMMAS FOR PROGRAM EQUIVALENCE</td>
</tr>
<tr>
<td></td>
<td>B.1 Properties of Imported Variables</td>
</tr>
<tr>
<td></td>
<td>B.2 Properties of Expression Evaluation</td>
</tr>
<tr>
<td></td>
<td>B.3 Properties of Remaining Execution</td>
</tr>
<tr>
<td>C</td>
<td>THE PROOF FOR EQUIVALENT TERMINATING COMPUTATION THEOREM</td>
</tr>
<tr>
<td>D</td>
<td>THE PROOF FOR EQUIVALENT TERMINATING BEHAVIOR THEOREM</td>
</tr>
<tr>
<td>E</td>
<td>THE PROOF FOR THE SAME I/O THEOREM</td>
</tr>
</tbody>
</table>
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Automatically Generated Mappings without Annotation</td>
<td>137</td>
</tr>
<tr>
<td>6.2</td>
<td>Automatically Detected Update Types without Annotation</td>
<td>138</td>
</tr>
<tr>
<td>6.3</td>
<td>Automatically Generated Mappings with Annotation</td>
<td>140</td>
</tr>
<tr>
<td>6.4</td>
<td>Automatically Detected Update Types with Annotation</td>
<td>141</td>
</tr>
<tr>
<td>6.5</td>
<td>PDG Matching Time for Vsftpd Updates without Annotation</td>
<td>143</td>
</tr>
<tr>
<td>6.6</td>
<td>Alias Analysis Time for Vsftpd</td>
<td>145</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Comparison of Targeted Programs, Overhead, Safety Guarantee, Quiescence Requirement</td>
<td>8</td>
</tr>
<tr>
<td>3.1 Six Cases of Formalized General New Program Behavior- Part1</td>
<td>26</td>
</tr>
<tr>
<td>3.2 Six Cases of Formalized General New Program Behavior - Part2</td>
<td>27</td>
</tr>
<tr>
<td>4.1 Abstract Syntax</td>
<td>30</td>
</tr>
<tr>
<td>4.2 Values, Types and Domains</td>
<td>31</td>
</tr>
<tr>
<td>4.3 Elements of an Execution State</td>
<td>32</td>
</tr>
<tr>
<td>4.4 Contextual Semantic Rule</td>
<td>33</td>
</tr>
<tr>
<td>4.5 SOS Rules for Expressions</td>
<td>34</td>
</tr>
<tr>
<td>4.6 SOS Rules for Assignment, If, and While Statements</td>
<td>35</td>
</tr>
<tr>
<td>4.7 SOS Rules for Input/Output Statements</td>
<td>36</td>
</tr>
<tr>
<td>5.1 Statistics of Classified Real World Software Update - Part1</td>
<td>78</td>
</tr>
<tr>
<td>5.2 Statistics of Classified Real World Software Update - Part2</td>
<td>79</td>
</tr>
<tr>
<td>5.3 Required Assumptions for Real World Backward Compatible Update Classes</td>
<td>80</td>
</tr>
<tr>
<td>5.4 Specializing New Configuration Variables</td>
<td>81</td>
</tr>
<tr>
<td>5.5 Enumeration Type Extension</td>
<td>82</td>
</tr>
<tr>
<td>5.6 Exit on Error</td>
<td>83</td>
</tr>
<tr>
<td>5.7 Missing Initialization</td>
<td>84</td>
</tr>
<tr>
<td>6.1 Two Versions of a Program Fragment</td>
<td>108</td>
</tr>
<tr>
<td>6.2 SitBack Architecture</td>
<td>118</td>
</tr>
<tr>
<td>6.3 An Example of Anti-Control Edges</td>
<td>124</td>
</tr>
<tr>
<td>6.4 Statistics of Updated Functions with At Least One Forward Jump of the Particular Type</td>
<td>126</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>6.5 Additional Formal Parameters</td>
<td>131</td>
</tr>
<tr>
<td>6.6 Annotation Categories</td>
<td>139</td>
</tr>
<tr>
<td>6.7 Plot of PDG Matching Time for Vsftpd Updates</td>
<td>143</td>
</tr>
<tr>
<td>6.8 An Example of Function Pointer Usage</td>
<td>144</td>
</tr>
<tr>
<td>6.9 Plot of Log2-Scale Alias Analysis Time for Vsftpd</td>
<td>145</td>
</tr>
<tr>
<td>6.10 Statistics of Automated State Mapping for Real World Software Update</td>
<td>147</td>
</tr>
<tr>
<td>6.11 Statistics of Base Programs and Auto-map for DSU</td>
<td>148</td>
</tr>
<tr>
<td>6.12 Some Mapped Changes in Sshd 3.6.1p1 - p2</td>
<td>149</td>
</tr>
<tr>
<td>6.13 Some Missed Changes in Sshd 3.6.1p1 - p2</td>
<td>150</td>
</tr>
<tr>
<td>6.14 Some Mapped Changes in Vsftpd 2.0.0 - 2.0.1</td>
<td>151</td>
</tr>
<tr>
<td>6.15 Some Missed Changes in Vsftpd 2.0.0 - 2.0.1</td>
<td>152</td>
</tr>
<tr>
<td>6.16 Some Mapped Changes in Cherokee 0.4.2b1 - b2</td>
<td>154</td>
</tr>
<tr>
<td>6.17 Missed Change in Cherokee 0.4.2b1 - b2</td>
<td>154</td>
</tr>
<tr>
<td>A.1 Typing Rules</td>
<td>168</td>
</tr>
<tr>
<td>B.1 Extended SOS Rule for Expressions</td>
<td>171</td>
</tr>
</tbody>
</table>
The ability of running a program without interruption is a highly desirable business and technical need. In practice, running programs are usually stopped to be updated for various reasons like bug fixes or functionality enhancement. However, the cost of system program downtime for software update is significant. In fact, it is estimated that the average cost of one service interruption for large businesses is around 1.5 million dollars [80, 9, 44, 60]. Even those businesses that could not afford business interruption, still suffer loss due to the IT interruption, including air carriers [92], healthcare service [70, 72], spacecraft [24, 73], financial service [30, 33], telecom service [16]. To address the need for non-interrupted operations, researchers proposed dynamic software update (DSU) to allow a program to be updated in the middle of its execution. DSU is useful for high-availability applications that cannot afford the downtime incurred by offline updates [50] or a long running program whose users are inconvenienced by updates. DSU has been an active area of research [50, 66, 48, 59, 15, 28, 47, 64, 36, 68, 37]. Researchers proposed DSU for operating systems [59, 15, 85] as well as general purpose programs [47, 67, 90]; researchers also proposed DSU for unmanaged programming languages (e.g., C) [74] and managed languages (e.g., Java) [90]; there are also DSU systems for standalone programs [66] or distributed systems [14].

Most of the published DSU work emphasizes the update mechanism that implements a state mapping which maps the execution state of an old version of the program to that of a new version. However, DSU safety has not yet been satisfactorily studied. Existing studies on DSU safety are lacking in one way or another: high-level studies
are concerned with change management for system components [56, 23] and lower-level studies typically require significant programmer annotations [46, 65, 95] or have a restricted class of applications to which they apply (e.g., controller systems [77]). There is no systematic study on the state mapping problem in DSU. By “systematic study”, we mean understanding the conditions under which a state mapping can be constructed and devising algorithms for constructing such mapping automatically when possible. In this dissertation we make an important step towards a more general understanding by considering the state mapping problem for server programs with a large user base. Such programs are characterized by the impracticality of updating client code which, we argue in this dissertation, restricts the kinds of updates that can be applied at runtime. Even though the state mapping problem is undecidable in general [42], for the class of programs we are considering, we show that it is possible to construct state mappings automatically or semi-automatically for backward compatible update.

As to our formal treatment of the state mapping problem, we consider the safety of DSU when applied to possibly non-terminating programs interacting with an environment that is not necessarily updated. For such updates, the new program must be able to interact with the old environment, which means that it should be, in some sense, backward compatible with the old program. A strict definition of backward compatibility would require the new version to exhibit the same I/O behavior as the old version (observational equivalence). However, it should be immediately clear that a more nuanced definition is needed because observational-equivalence does not allow changes such as bug fixes, new functionalities, or usability improvement (e.g., improved user messages). Allowing for such differences would be needed in any practical definition of backward compatibility. One contribution of this dissertation is a detailed definition of backward compatibility.
In general, determining backward compatibility between two different program versions requires solving the *semantic equivalence* problem which has been extensively studied [52, 39, 20, 57, 91, 55, 58, 63]. Unfortunately, existing results are lacking in one or more aspects which rules out retrofitting them for our setting. Existing work on program equivalence does not allow us to express that a point in the middle of a loop execution of one program *corresponds* to a point in the middle of a loop execution of another program. The ability to express such correspondences is desirable for DSU. Besides, existing formulations of the program equivalence problem either do not use formal semantics [52, 21, 55], only apply to terminating programs [20, 57], severely restrict the programming model [39, 91, 55], or rely on model checking [58, 63, 57, 45] (which is not appropriate for non-terminating programs with infinite states). Our goal for program equivalence is to establish compile-time conditions ensuring that two programs have the same I/O behavior in *all* executions. In this dissertation, we propose a framework of syntactic conditions for program equivalence. This is different from much of the literature on program equivalence which only guarantees same behavior in terminating executions. A detailed discussion of related work is in Chapter 2.

We present a study of real world program evolution to understand real world update characteristics. The study of real world program evolution also helps identifying backward compatible update patterns. Our study involves 34 consecutive updates of three widely used programs, namely vsftpd [12], sshd [11] and icecast [10]. We identify and summarize classes of backward compatible updates. Based on the study of backward compatible updates, we formalize and prove those summarized update classes to be backward compatible.

To show the applicability of our methodology of state mapping, we developed a tool that automates the construction of (partial) state mappings for backward com-
patible DSU of real world programs. We show a general approach of automating
the construction of a state mapping for real world programs, and present a new tool
SitBack (Static Inference Tool for BACKward compatibility) that is able to automatic-
cally generate mappings with limited user annotation for changes involving bug fixes,
type relaxation, function generalization, changes to log functions, code reordering
and variable renaming. SitBack determine the conditions under which the program
dependence graph of the new updated application is backward compatible with the
program dependence graph of the old application and infers from these conditions how
the state of the old application is to be mapped. SitBack tries to construct a state
mapping by matching program dependence graphs (PDG) of changed functions from
an old version of a program to a new version of the program. A PDG captures control
dependence and data dependence in a function [52]. However, real world programs
usually include jumps, which is not considered in the language model for PDG [52].
We extended PDG for a particular type of jumps observed in real world program evo-
lution in order to detect equivalence for programs with that particular type of jumps.
We show SitBack to be an effective state mapping tool by a detailed evaluation. In
a case study, we have shown our approach to be effective by testing it on 5.5 years
worth of updates of the vsftpd secure file transfer software and our system was able
to generate mappings for 39 – 100\% of the changed functions with user annotation
and for 17 – 75\% of the changed functions without user annotation. The system also
generates mappings for functions that did not change textually but that have to be
supported with non-trivial mappings due to changes in data structures that they use.
It is important to emphasize here that falling short of a full automation of the state
mapping problem is not a limitation of our approach, but an inherent limitation of
the problem.
1.1 Contributions

This dissertation has three main contributions: (1) Identifying and defining backward compatibility as a correctness criterion for DSU of server applications whose clients are not updated; (2) a study of real world program evolution to identify patterns of code update that result in backward compatible behavior; and (3) the development of a tool that assists developers to obtain a state mapping for backward compatible update.

1. The first contribution includes two parts:

   (a) A formal definition of backward compatibility. Backward compatibility is a general assumption guiding our automatic generation of state mapping for interactive programs with an established user base that could not be updated.

   (b) A framework of equivalence for both terminating and non-terminating interactive programs. The framework of program equivalence is the core of our formalizing various update classes.

2. The second contribution: A classification of commonly encountered provably backward compatible changes with a general approach to dealing with them.

3. The third contribution is composed of two parts:

   (a) An algorithm for calculating a state mapping for dynamic software updates.

   (b) The first implementation of a software tool to calculate state mapping with well defined semantic guarantees.
1.2 Organization

The rest of the dissertation is organized as follows. Related work is discussed in Chapter 2. Chapter 3 proposes backward compatibility as a general correctness condition for dynamic software updates and identifies categories of backward compatible program behavior. Chapter 4 presents a framework of equivalence for interactive programs which is the core of our formal study of DSU safety. Chapter 5 presents our study of real world program evolution and formalizes provably backward compatible update classes corresponding to categories of behaviors described in Chapter 3. We introduce our implementation of our state mapping tool in Chapter 6. We conclude this dissertation and propose future work in Chapter 7.
Chapter 2

RELATED WORK

This chapter includes the discussion of related work on (1) general DSU systems, (2) DSU correctness criteria, and (3) automatic calculation of state mapping for DSU.

2.1 General Dynamic Software Update System

We discuss DSU systems on various aspects, including targeted programs (programming language, general application or OS, program concurrency), DSU overhead, safety guarantee, and quiescence requirement. All listed DSU systems are (or were) either used in practice or are known to be able to update real world programs. The summary of our related work is shown in Fig 2.1.

1. (Targeted Programs) We discuss the targeted programs of listed DSU systems. The targeted programs of a DSU system decide the design, applicability and evaluation of the DSU system.

- (Targeted Program Instrumentation) All listed DSU systems require instrumentation of targeted programs except Ksplice. This is because Ksplice works with the compiled binary of target programs. Due to difference in DSU mechanism, DSU systems require different types of program instrumentation. For example, Ginseng and UpStare the program instrumentation involves function redirection (to update changed functions) and type wrapping. Ginseng does type wrapping by relating each type with a version number. In this way, Ginseng is able to transform values of an old type to an updated type. UpStare does type wrapping by grouping local
<table>
<thead>
<tr>
<th>DSU System</th>
<th>Targeted Programs</th>
<th>Overhead</th>
<th>Safety</th>
<th>Quiescence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Language</td>
<td>Category</td>
<td>Concurrency</td>
<td>(Type, Range)</td>
</tr>
<tr>
<td>K42 [18]</td>
<td>C++</td>
<td>OS</td>
<td>Process</td>
<td>ObjCreat, [0.73%, 12%]</td>
</tr>
<tr>
<td>OPUS [36]</td>
<td>C</td>
<td>App</td>
<td>Thread</td>
<td>PatchTime, [39.25ms, 81.44ms]</td>
</tr>
<tr>
<td>DynAMOS [69]</td>
<td>C/Asm</td>
<td>OS</td>
<td>Thread, Process</td>
<td>SysCallTime, [0.83%, 43.92%]</td>
</tr>
<tr>
<td>LOCUS [27], POLUS [29]</td>
<td>C</td>
<td>App</td>
<td>Thread</td>
<td>Time, [-0.092%, 0.525%]</td>
</tr>
<tr>
<td>GinSeng [74]</td>
<td>C</td>
<td>App</td>
<td>Thread</td>
<td>Time, [-0.19%, 6.71%]</td>
</tr>
<tr>
<td>Kaplice [15]</td>
<td>C/Asm</td>
<td>OS</td>
<td>Thread</td>
<td>“No measurable performance impact”</td>
</tr>
<tr>
<td>JVolve [90]</td>
<td>Java</td>
<td>App</td>
<td>Thread</td>
<td>Time, [-29.63%, -13.45%]</td>
</tr>
<tr>
<td>UpStare [67, 66]</td>
<td>C</td>
<td>App</td>
<td>Thread, Process</td>
<td>Time, [16.0%, 96.4%]</td>
</tr>
<tr>
<td>Kitsune [47]</td>
<td>C</td>
<td>App</td>
<td>Thread</td>
<td>Time, [-2.18%, 2.35%]</td>
</tr>
<tr>
<td>DynSec [81]</td>
<td>C</td>
<td>App</td>
<td>Thread</td>
<td>Time, [7%, 27%]</td>
</tr>
<tr>
<td>Proteos [37]</td>
<td>C</td>
<td>OS</td>
<td>Thread, Process</td>
<td>“no noticeable time overhead”</td>
</tr>
<tr>
<td>Rubah [84]</td>
<td>Java</td>
<td>App</td>
<td>Thread</td>
<td>Time, [-1.0%, 2.5%]</td>
</tr>
</tbody>
</table>

Figure 2.1: Comparison of Targeted Programs, Overhead, Safety Guarantee, Quiescence Requirement

variables of a function into a structure so that it is able to reconstruct a stackframe when there are changes of local variables of the function. UpStare also needs to insert program points (labels) into the original programs to identify where the old program’s execution left off and where the new program shall start from. Kitsune does not need function redirection and type wrapping, however, Kitsune requires adding branching and program points in original programs to migrate new program’s execution to a point where the old program’s execution left off. Kitsune also needs to register local variables for migrating the global/local variables of an old program to those of a new version of the program.
• **(Programming Language of Targeted Programs)** Ten DSU systems were engineered for targeted programs in C or C++ and the other two are for Java. The two DSU systems for Java are partially due to the recent wide use of Java in server (long-running) program implementation like Hadoop [2], HBase [3], Hive [1], just to name a few. There are several differences among DSU systems for C/C++ or Java. For Java, there is a virtual machine which tracks all the memory accessed by a running program including the stack, the heap and the global variables. The access modifier (private, protected) in Java and C++ add complexity for transforming an object of an old class to that of a new class. Features like dynamic binding of Java or C++ does not add complexity to a DSU system as long as type safety for classes is guaranteed.

• **(Targeted Program Type)** Eight DSU systems are designed for general application update. K42 and Proteos are two OS engineered with DSU in mind. A DSU system targeted for OS programs usually has more access to system resource like process privilege, kernel symbol table. It is worth mentioning that Ksplice is the only known widely deployed DSU system in real world. This is partly because Ksplice only targets small OS security patches and causes no measurable performance impact to a running OS due to no program instrumentation.

• **(Targeted Program Concurrency)** All of the DSU systems support updating multi-threaded programs. And only K42 and Proteos provide DSU for multi-process programs due to OS’s required support of multi-process programs. Proteos supports dynamic updates with the presence of running multi process programs by adding redirection to the IPC implementation.
2. **(Overhead)** The overhead refers to extra used memory or time spent on program execution due to the program instrumentation by a DSU system. When a DSU system requires program instrumentation, performance overhead is unavoidable [19]. We collect the overhead measurement for all the DSU systems from their respective publications if any. For Ksplice, we contacted the author and learnt that “there is no measurable performance impact”. The reported overhead is measured in different methodology; hence they are not directly comparable. For example, the overhead measurement for K42 is on the time of object creation, the measurement of overhead for OPUS only includes the time to apply a security patch, the measurement of DyAMOS overhead is on the time of system call. The throughput overhead of LOCUS/POLUS, Ginseng, JVolve, UpStare, and DynSec is related to specific programs in evaluation (e.g., JavaEmailServer for JVolve, PostgreSQL for Upstare). Proteos is evaluated using benchmark SPEC CPU 2006 (CPU-intensive) [6] and sdtools (syscall-intensive). Instrumentation overhead may not affect throughput depending on the combination of the updated part of programs and the I/O hardware. JVolve and Rubah rely on the adaptive Just-In-Time compiler to optimize the program performance so that some of the time overhead is negative. Kitsune does program instrumentation without function redirection and type wrapping, but induces negative time overhead. It is unclear how Kitsune is able to induce negative overhead. By limiting the update points, UpStare could reduce overhead to a level that is similar to what Kitsune incurs. Ginseng also induces negative overhead. From a private correspondence, Ginseng’s author mentioned that the negative overhead is partly because “both memory footprint and throughput rate are heavily influenced by other factors (e.g., size of code in libraries and
I/O throughput) that trump the memory and performance overhead introduced by DSU”.

3. **(Safety)** We consider three types of safety in our comparison, namely type safety, representation safety, and thread safety proposed in [67]. We don’t include transaction safety [67] which means that “some sections of code are denoted as transactions and are specified by the user to execute completely in the old version or completely in the new version” [67]. This is because that, despite of the difference in DSU system mechanism, DSU systems could ensure transaction safety by “conservatively limiting the update points” based on given transaction annotation by developers at compile time [67]. Type safety means “no old version of code should be executed on a newer version of datatype representation” [67] and vice versa. Representation safety is of two parts. The first part is *state representation consistency*, which means that “at no time the executing application expects different representation of state (such as global variables or the stack-frame contents)” [67]. The second part is *program representation consistency*, which means that, after the update, only new code is executed. Representation consistency is different from type safety in that an old program is allowed to be executed after the update as long as that old program only access old types. Thread safety means that the first two kinds of safety is guaranteed for multi-threaded programs.

- **(Type Safety)** All listed DSU systems except POLUS. POLUS only ensures type safety for global variables. OPUS, Ksplice and DynSec ensures type safety by disallowing data type changes.

- **(Representation Safety)** UpStare and Kitsune ensure representation consistency because both DSU systems update the running stack of a pro-
gram. UpStare and Kitsune adopt two different ways of stack reconstruction. UpStare constructs the stack for a new version of a program by unrolling the stack of a running old version of a program and then reconstructing the stack of the new version in the reverse order of unrolling. UpStare also updates the program counter. Instead of manipulating the stack of a running old program, Kitsune starts a new program execution from scratch and migrates the new program execution to where the old program execution left off. Before the update, Kitsune needs to instrument an old program and a new program for several things. One is to add update points with distinct names and to add conditional in the new program to allow skipping the normal execution path when migrating a new program execution. Another kind of instrumentation is to add function calls in an old program to register global/local variables and to add function calls in a new program to restore global/local variables values. Notice that the update points in the two programs are corresponding. Kitsune first call setjmp in its runtime driver. Upon an update (given the update point name), Kitsune copies registered local variables (for data migration) to the heap and then calls longjmp to start the new program execution from scratch and migrates the execution to the corresponding update point where the old version left off. Kitsune speeds up the migration of the new program execution by skipping the new program’s normal execution path and checking the update point to guide the progression of the execution. By the program instrumentation, Kitsune registers global/local variables (name, address) that need to be migrated in instrumented functions marked as a constructor. At the entrance of the execution of the main function in the new program, Kitsune migrates the global variables.
In the execution of each subroutine, Kitsune migrates local variables if needed before proceeding to the next statement.

- **(Thread Safety)** All the DSU systems ensure their respective thread safety in different ways. For example, UpStare “adapts an algorithm that blocks all threads in heterogenous checkpointing for multi-threaded applications [54]” [66]; Ginseng relies on static analysis to reason about the safe update points for all threads to be blocked [74]; Ksplice relies on a special kernel instruction to force all threads to stop [15].

4. **(Quiescence Requirement)** Quiescence means that the DSU system requires the updated code to be not active on the stack at the time of updating. This is usually needed for DSU systems that do not update the running stack of a program. UpStare and Kitsune do not require quiescence because the two DSU systems reconstruct the stack when updating a program. JVolve, Rubah and Proteos do not require quiescence either. However, the three DSU systems require developers to ensure type safety for the relaxed quiescence requirement. The other DSU systems do require quiescence in different ways. Due to the mechanism of a DSU system, quiescence may not be possible for DSU systems like Ginseng with the presence of certain program structures, for example, a long running event-handling loop. Ginseng proposed to extract a long running loop to a recursive function and does static analysis reasoning that the beginning of the extracted loop is quiescence point good for update.

2.2 DSU Correctness Criteria

The related work on the DSU correction criteria is of two parts. We discuss related work on DSU semantic correctness and program equivalence in order.
Existing studies on DSU semantic correctness could be roughly divided into high level studies and low level ones. There are a few studies on high level DSU correctness. Gupta et al. [42] proposed a general definition of update correctness, that an “online software change” is valid if, after the update, the program is guaranteed to reach a reachable state within a finite amount of time. Gupta et al. [42] also showed that the update correctness problem is undecidable. In [56], Kramer and Magee defined DSU correctness that the updated system shall “operate as normal instead of progressing to an error state”. In [23], Bloom and Day proposed DSU correctness which allows functionality extension that could not produce past behavior. This is probably because Bloom and Day considered updated environment. Panzica La Manna et al. [77] presented high level correctness only considering scenario-based specifications (describing allowed sequences of events) for finite state controller systems instead of general programs.

There are also studies on low level DSU safety. Hayden et al. [46] discussed DSU correctness and concluded that there is only client-oriented correctness. Zhang et al. [95] asked the developers to ensure DSU correctness. Magill et al. [65] proposed automatic state mapping between two corresponding functions by comparing function states after test executions of the two functions. It is not clear what correctness is ensured by the obtained state mapping in [65].

We next discuss existing work on program equivalence because backward compatibility is closely related to program equivalence. There is a rich literature on program equivalence and we only discuss those most related work. Horwitz et al. [52] proposed program equivalence by checking isomorphic program dependence graphs (PDG). In [39], Godlin and Strichman have a structured study of program equivalence. However, Godlin and Strichman [39] restricted the equivalence by transforming every loop into a recursive function and requiring functions with a same name to be
equivalent. Such loop transformations make it impossible to detect equivalence with the presence of loop fission, loop fusion and loop invariant code motion.

Another class of work on program equivalence is to leverage the power of logic solver. The basic idea is to interpret a program into a formula and then use an on-the-shelf theorem prover to identify if two formula are equivalent in any case [57, 39, 45, 91, 55, 58, 63, 20, 61]. The theorem prover is powerful in finding non structure preserving equivalent semantic rewriting. For example, “x += 2” will be found equivalent to “x +=1; x+=1;”. However, the theorem prover is not appropriate for non terminating execution where the number of models is infinite.

2.3 Automatic Generation of State Mapping

We found few related work trying to solve the state mapping problem directly [65, 79, 38]. As is mentioned in the previous section, existing work focuses on the verification of the correctness of a given state mapping [46, 95]. Magill [65] proposed a state mapping by comparing the execution state of two corresponding functions from a set of test executions. But the obtained state mapping depends on the chosen execution set and it is unclear how to select a set of executions to discover all the differences between two programs or functions. Consequently it is unclear how such state mapping ensures DSU correctness. Giuffrida et al. [38] proposed a solution of automating the construction of a struct type transformer for struct type changes like field movement, type weakening or strengthening (e.g., int to long, array size increase or decrease), and new fields. However, Giuffrida et al. [38] requires developer’s help to do state transfer for any program code change. Partush and Yahav [79] proposed a methodology to verify program state matching by identifying equivalence and difference of corresponding variables in an interleaving execution of a program and its
patched version. Partush and Yahav [79] assumed variable matching by name or a developer.

Our state mapping methodology is based on the use of PDG [52]. A PDG captures control and data dependence (flow, def-order) in a function. There are several extension of PDG [31, 86, 17]. In [21], Binkley et al. extended the result of program equivalence from PDG isomorphism to multiple procedure programs. In [51], Horwitz et al. extended program dependence for programs with pointers. In [17], Ball et al. extended PDG in order to create correct slicing with the presence of jump statement in a function. As to jump, it is unclear how to extend PDG for programs with arbitrary jump. However, in our study of real world program evolution, we do not observe arbitrary jump. Instead, we only observe one particular type of jumps where the jump is toward the end of a function that post-dominates the entry of the function. We have extension of PDG for this particular type of jump so that it is possible to check program equivalence from isomorphism of our PDG extension.

There is interpretation of program dependence in Horwitz’s PDG using denotational semantics [26]. However, it is not clear how to use the denotational semantics of program dependence to capture program equivalence in the middle of a possibly non-terminating execution.
We argue that backward compatibility is required for dynamic program updates when the environment of the program execution does not necessarily update. Then we present a formal detailed discussion of backward compatibility. The organization of this chapter is as follows. We first formally define a program, an execution, and a program specification. Then we formalize the hybrid execution in DSU. We finally propose the formal definition of backward compatible DSU.

3.1 Program, Execution and Specification

A program is designed to satisfy a specification. A specification can be explicitly provided or implicitly defined by the behavior of a program. A program interacts with its environment by receiving inputs and sending outputs. In this chapter we introduce enough of a computing model to describe the input/output behavior of programs; In the next chapter we introduce a specific programming language to reason about specific software updates.

An execution of a program consists of a sequence of steps from a finite set of steps, $S = S_{in} \cup S_{internal} \cup S_{out} \cup \{halt\}$. A step of a program can either be an input step in which an input is received, an internal step in which the state of the program is modified, an output step in which an output is produced, or a halt.

We make a distinction between internal states of a program and external states (e.g., application settings) of the local environment in which the program executes. Such external state can include the state of a file system that a program can access; we include both as part of the program state. The state of a program is an element of
a set $\mathcal{M} \times \mathcal{I}$, where the set $\mathcal{M} = \mathcal{M}_{\text{int}} \times \mathcal{M}_{\text{ext}}$, $\mathcal{M}_{\text{int}} = \prod_{k=0}^{n_{\text{int}}} V_k$ is a cartesian product of $n_{\text{int}}$ sets of values, one for each internal memory location, and $\mathcal{M}_{\text{ext}} = \prod_{k=0}^{n_{\text{ext}}} V_k$ is a cartesian product of $n_{\text{ext}}$ sets of values, one for each external location. The input value last received is an element of the set $\mathcal{I}$ of input values.

A program executes in an 
\textit{execution environment}. An execution environment $(M_{\text{ext}0}, I)$ specifies an initial value for the external program state $M_{\text{ext}0}$ and a possibly infinite sequence of input values $I$. The input sequence is assumed to be produced by \textit{users} that we do not model explicitly.

A step of a program $P$ is a mapping that specifies the next program state and the next step to execute. For an internal step $s_{\text{internal}} \in S_{\text{internal}}$, the mapping is $s_{\text{internal}} : \mathcal{M} \times \mathcal{I} \mapsto S \times \mathcal{M} \times \{\perp\}$, which specifies the next step and how the state is modified. The internal steps clear input in the state if any. For an output step $s_{\text{out}} \in S_{\text{out}}$, the mapping $s_{\text{out}} : \mathcal{M} \mapsto S \times \mathcal{O}$ which specifies the next step to execute and the output value produced. $\mathcal{O}$ is the set of output values produced by a program.

An input step $s_{\text{in}} \in S_{\text{in}}$ is simply an element of $S \times \mathcal{I}$ and specifies the next step to execute and the input obtained from an environment. (We simply write $s_{\text{in}}(\cdot)$ to denote the next step and the input received.) Because the input value is received by a program, we do not restrict the next step to execute. We allow the input value to be ignored by the program by two consecutive input steps. When the step is \textit{halt}, there is no further action as if \textit{halt} were mapped to itself.

\textbf{Definition 1. (Program)} A program $P$ is a tuple $(S, \mathcal{M}, M_{\text{int}0}, s_0, \mathcal{I}, \mathcal{O})$, where $S$ is the set of steps as defined above, $\mathcal{M}$ is the set of program states, $M_{\text{int}0}$ is the initial internal state, $s_0$ is the initial step, and $\mathcal{I}$ and $\mathcal{O}$ are disjoint sets of input and output values.
We do not include the initial external state $M_{ext0}$ in the program definition; we include it in the execution environment of $P$.

**Definition 2. (Execution)** An execution of a program $P = (S, M, M_{int0}, s_0, I, O)$ in execution environment $(M_{ext0}, I)$, where $I$ is a possibly infinite sequence of input values from $I$, is a sequence of configurations $C$ from the infinite set $\{(M, s, i, I_r, IO)\}$. A configuration $c$ has the form $c = (M, s, i, I_r, IO)$, where $M$ is a state, $s$ is a step, $i$ is the last input received, $I_r$ is a sequence of remaining input values and $IO$ is the input/output sequence produced so far. The $k$th configuration $c_k$ in an execution is obtained from the $(k-1)$th configuration $c_{k-1} = (M, s, i, I_r, IO)$ where $s \neq \text{halt}$ in one of the following cases:

1. The first configuration $c_0$ is of the form $(M_0, s_0, \bot, I, \emptyset)$, where $M_0 = (M_{int0}, M_{ext0})$;

2. $s \in S_{\text{internal}} : c_k = (M', s', \bot, I_r, IO)$, where $(s', M', \bot) = s(M, i)$;

3. $s \in S_{\text{in}}$ and the remaining inputs $I_r$ is not empty: $c_k = (M, s', \text{head}(I_r), \text{tail}(I_r), IO \cdot \text{head}(I_r))$ where $(s', \text{head}(I_r)) = s(I_r)$;

4. $s \in S_{\text{in}}$ and the remaining inputs $I_r$ is empty: $c_k = c_{k-1}$;

5. $s \in S_{\text{out}} : c_k = (M, s', i, I_r, IO \cdot o')$, where $(s', o') = s(M)$;

In the definition, $\text{head}(I)$ denotes the head (leftmost) element in the sequence $I$ and $\text{tail}(I)$ denotes the remaining sequence without the head. The input value in $i$ is either consumed by the next internal step or updated by another input from the next input step. Execution is stuck if an input step is attempted in state in which there are no remaining inputs. In what follows, we include the execution environment in the execution and we abuse notation to say $(M_{ext0}, I, C)$ is an execution of a program $P$. 

19
**Specification**  We consider specifications that define the input/output behavior of programs. Specifications are not concerned with how fast an output is produced or about the internal state of the program.

**Definition 3. (Specification)** Given a set $\mathcal{M}_{\text{ext}}$ of external states, a set $\text{seq}(\mathcal{I})$ of input sequences, and a set $\text{seq}(\mathcal{I} \cup \mathcal{O})$ of I/O sequences, specification $\Sigma$ is a predicate: $\mathcal{M}_{\text{ext}} \times \text{seq}(\mathcal{I}) \times \text{seq}(\mathcal{I} \cup \mathcal{O}) \to \{\text{true}, \text{false}\}$.

We define the I/O sequence of a sequence of configurations $C$ to be a sequence $IO(C)$ of values from $\mathcal{I} \cup \mathcal{O}$ such that every finite prefix of $IO(C)$ is the I/O sequence of some configuration $c \in C$ and every I/O sequence of a configuration $c \in C$ is a finite prefix of $IO(C)$.

An execution $(M_{\text{ext}0}, I, C)$ of program $P$ satisfies a specification $\Sigma$ if $\Sigma(M_{\text{ext}0}, I, IO(C)) = \text{true}$. A specification distinguishes executions into those that satisfy the specification and those that do not.

A specification defines the external behavior of a program that is observed by a user. The input sequence and I/O sequence are obviously part of external behavior. We also include $\mathcal{M}_{\text{ext}}$ in specification domain because a user can have information about the external state. For example, a user who has data stored in the file system considers the program’s refusal to access the stored data a violation of the service specification; this is not the case if the user has no stored data.

### 3.2 Hybrid Execution, State Mapping and Backward Compatibility

DSU is a process of updating software while it is running. This results in a hybrid execution in which part of the execution is that of an old program and part of the execution is for a new program.
State mapping is a function $\delta$ mapping an internal state and a non-halt step of one program $P$ to an internal state and a step of another program $P'$, $\delta : \mathcal{M}_{int}^P \times (\mathcal{S}^P \setminus \{halt\}) \mapsto \mathcal{M}_{int}^{P'} \times \mathcal{S}^{P'}$. The external state is not mapped because the environment is not necessarily updated. In addition, we cannot change input and output that already occurred and that I/O must be part of the hybrid execution.

**Definition 4. (Hybrid Execution)** A hybrid execution $(M_{ext}, I, C_P; C_{P'})$, produced by DSU using state mapping $\delta$ from program $P$ to program $P'$, is an execution $(M_{ext}, I, C_P)$ of $P$ concatenated with an execution $(M'_{ext}, I', C_{P'})$ of $P'$ where the first configuration $c_{P'} = ((M'_{int}, M'_{ext}), s', i', I', IO')$ in $C_{P'}$ is obtained by applying the state mapping to the last configuration $c_P = ((M_{int}, M_{ext}), s(\neq \text{halt}), i, I_r, IO)$ in $C_P$ as follows:

- $(M'_{int}, s') = \delta(M_{int}, s)$;
- $(i' = i) \land (I_r' = I_r) \land (IO' = IO) \land (M_{ext} \subseteq M'_{ext})$.

In this dissertation, we consider updates in which the environment is not necessarily updated. It follows that in order for a hybrid execution to be meaningful, the new program should provide functionality expected by both old and new users of the system.

In practice, specifications are not explicitly available. Instead, a program is its own specification. This means that the specification that a program satisfies can only be inferred by the external behavior of the program. Bug fixes create a dilemma for dynamic software updates. When a program has a bug, its external behavior does not captures its implicit specification and the update will change the behavior of the program. In what follows, we first discuss what flexibility we can afford for a backward compatible update and then we give formal definitions of backward compatibility and state our assumptions for allowing bug fixes.
We consider a hybrid execution starting from a program \( P = (S, \mathcal{M}_{int} \times \mathcal{M}_{ext}, M_{int_0}, s_0, \mathcal{I}, \mathcal{O}) \) and being updated to a program \( P' = (S', \mathcal{M}'_{int} \times \mathcal{M}'_{ext}, M'_{int_0}, s'_0, \mathcal{I'}, \mathcal{O'}) \). We examine how the two programs should be related for a meaningful hybrid execution.

1. (Inputs) Input set \( \mathcal{I'} \) of \( P' \) should be a superset of that \( \mathcal{I} \) of \( P \) to allow for old users to interact with \( P' \) after the update. It is possible to allow for new input values in \( \mathcal{I'} \) to accommodate new functionality under the assumption that old users do not generate new input values. Such new input values should be expected to produce erroneous output by old users as they are not part of \( P \)'s specification.

2. (Outputs) Output produced by \( P' \) should be identical to output produced by \( P \) if all the input in an execution comes from the input set of \( P \). This is needed to ensure that interactions between old users and the program \( P' \) can make sense from the perspective of old users. This is true in the case that the update does not involve a bug fix, but what should be done if the update indeed involves a bug fix and the output produced by the old program was not correct to start with? As far as syntax, a bug fix should not introduce new output values. As far as semantics, we should allow the bug fix to change what output is produced for a given input. We discuss this further under the bug fix heading. In summary, if we ignore bug fixes, the new program should behave as the old program when provided with input meant for the old program.

3. (Bugfix) Handling bug fixes is problematic. If the produced output already violates the fix, then there is no way for the hybrid execution to satisfy the implicit semantics of the program or the semantics of the new program. Some bug fixes can be handled. For example, a bug that causes a program to crash for some input can be fixed to allow the program to continue executing. Applying
the fix to a program that has not encountered the bug should not be problematic. Another case is when the program should terminate for some input sequence, but the old program does not terminate. A bug fix that allows the program to terminate should not present a semantic difficulty for old users.

In general, we assume that there are valid executions and invalid executions of the old program. I/O sequences produced in invalid executions are not in specification of the program. We assume that an invalid execution will lead to an error configuration not explicitly handled by the program developers. We do not expect the state mapping to change an error configuration into an non-error configuration just as static updating does not fix occurred errors. Besides, we do not attempt to determine if a particular configuration is an error configuration. Such determination is not possible in general and very hard in practice. We simply assume that the configuration at the time of the update is not an error configuration. (which is equivalent to assuming the existence of an oracle $J_P$ to determine if a particular configuration is erroneous, $J_P(C_P) = true$ if the configuration $C_P$ is not erroneous).

4. (New Functionality) New functionality is usually accompanied by new inputs / outputs and the expansion of external state. We assume that new functionality is independent of existing functionality in the sense that programs $P$ and $P'$ produce the same I/O sequence when receiving inputs in $\mathcal{I}$ only. We therefore assume all new inputs $\mathcal{I}' \setminus \mathcal{I}$ are introduced by new functionality.

Every external state of $P$ is part of some external state of program $P'$ because of the definition of the specification of $P$. We only consider expansion of the external state of $P$ for new functionalities in $P'$ where the expansion of external
state is independent of values in existing external state. One of the motivating examples is to add application settings for new program feature.

In light of the discussion above we give the following definition of backward compatibility in the absence of bug fixes.

**Definition 5. (Backward Compatible Hybrid Executions)** Let \( P = (S, M_{\text{int}} \times M_{\text{ext}}, s_0, \mathcal{I}, \mathcal{O}) \) be a program satisfying a specification \( \Sigma \). We say that a hybrid execution \((M_{\text{ext}}, I, C_P; C_P')\) from \( P \) to a program \( P' = (S', M'_{\text{int}} \times M'_{\text{ext}}, s'_0, \mathcal{I}', \mathcal{O}')\) is backward compatible with implicit specification of \( P \) if all of the following hold:

- The last configuration in \( C_P \) is not an error configuration, \( C_P = "C'"; (M, s', i, I_r, IO)" : \( J_P(C_P) = \text{true} \).

- The hybrid execution satisfies the specification \( \Sigma \) of \( P \),

\[ \Sigma(M_{\text{ext}}, I, IO(C_P; C_P')) = \text{true}; \]

- Inputs/outputs/external states of \( P \) are a subset of those of \( P' \) : \( \mathcal{I} \subseteq \mathcal{I}', \mathcal{O} \subseteq \mathcal{O}' \) and \( M_{\text{ext}} \subseteq M'_{\text{ext}} \);

If there is bug fix between programs \( P' \) and \( P \), we need to adapt Definition 5 to allow for some executions on input sequences from \( \mathcal{I} \) to violate the specification of \( P \). Above we identified two cases in which bug fixes are safe (replacing a response with no response or replacing a no response with a correct response without introducing new output values). We omit the definition.

We have the backward compatible updates by extending the definition of a backward compatible hybrid execution to all possible hybrid executions.

**Definition 6. (Backward Compatible Updates)** We say an updated program \( P' \) is backward compatible with a program \( P \) in configuration \( C \) if there is hybrid execu-
With the formal definition of backward compatibility, it is desirable to check what behavior changes of an updated program help ensure a safe update. Backward compatibility is essentially a relation between I/O sequences produced by an old program and those produced by an updated program. We summarized typical possibilities of the relation into six cases in Figure 3.1 and 3.2 by considering consequence of major update motivation (i.e., new functionality, bug fix and program perfective/preventive needs [5]). According to David Parnas [78], a program is updated to adapt to changing needs. In other words, program changes are to produce more or less or different output according to changing needs. These changes are captured by case 2, 3, 4, 5 and 6 in Figure 3.1 and 3.2. We also capture output-preserving changes which are most likely motivated by the program developer’s own needs (e.g., software maintainability), which is case 1 in Figure 3.1.

Furthermore, we find that an update is backward compatible if in every execution the new program behavior is one of the six cases in Figure 3.1 and 3.2. Cases 1 and 2 are obviously backward compatible because an old client is guaranteed to get old responses. Cases 3, 4, 5, and 6 are not obviously backward compatible. Unlike case 1 and 2, case 3, 4 and 5 are backward compatible under specific assumptions on program semantics while case 6 is different. Case 3 is backward compatible because we assume the change is either adding new functionality, or fixing a bug in which the old program hanged or crashed. Similarly, case 4 is backward compatible. Case 5 is backward compatible because different I/O interaction could express the same application semantics. For example, a greeting message could be changed from “hi”
<table>
<thead>
<tr>
<th>Case</th>
<th>Formal New Program Behavior</th>
</tr>
</thead>
</table>
| 1    | the old behavior including external state extension:  
\[ \Sigma_P \subseteq \Sigma_{P'} \]  
\[ \exists (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \rightarrow \text{val in } \Sigma_P \]  
and \( M_{ext} \subseteq M_{ext'} \) where \( \mathcal{I} = \mathcal{I}', \mathcal{O} = \mathcal{O}' \) and \( M_{ext} \subseteq M_{ext'} \) |
| 2    | the old behavior for old input and consuming inputs that are only from new clients:  
\[ \Sigma_P \subseteq \Sigma_{P'} \land \Sigma_{P'} \setminus \Sigma_P = \{(M_{ext}, \text{oneseq}(\mathcal{I}'), \text{oneseq}(\mathcal{I}' \cup \mathcal{O}')) \rightarrow \text{true} | \text{oneseq}(\mathcal{I}' \cup \mathcal{O}') \text{ includes at least one input in } (\mathcal{I}' \setminus \mathcal{I}) \neq \emptyset \} \]  
where \( \mathcal{I} \subseteq \mathcal{I}', \mathcal{O} \subseteq \mathcal{O}' \) and \( M_{ext} = M_{ext'} \) |
| 3    | producing more output while the old program terminates:  
\[ \Sigma_{P'} \setminus \Sigma_P = \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \rightarrow \text{false} | (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_f \neq \emptyset \} \]  
\[ \cup \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \cdot \text{oneseq}'(\mathcal{I} \cup \mathcal{O})) \rightarrow \text{true} | (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \cdot \text{oneseq}'(\mathcal{I} \cup \mathcal{O})) \in \Delta_t \neq \emptyset \} \]  
\[ \Sigma_P \setminus \Sigma_{P'} = \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \rightarrow \text{false} | (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_t \neq \emptyset \} \]  
\[ \cup \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \cdot \text{oneseq}'(\mathcal{I} \cup \mathcal{O})) \rightarrow \text{true} | (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \cdot \text{oneseq}'(\mathcal{I} \cup \mathcal{O})) \in \Delta_f \neq \emptyset \} \]  
where \( \mathcal{I} = \mathcal{I}', \mathcal{O} = \mathcal{O}' \) and \( M_{ext} = M_{ext'} \) |

Figure 3.1: Six Cases of Formalized General New Program Behavior - Part 1
<table>
<thead>
<tr>
<th>Case</th>
<th>Formal New Program Behavior</th>
</tr>
</thead>
</table>
| 4    | termination while the old program produces erroneous output:  
\[ \Sigma_P' \setminus \Sigma_P = \{(M_{\text{ext}}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{true} \} \]
\[ \cup \{(M_{\text{ext}}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \cdot \text{oneseq}'(\mathcal{I} \cup \mathcal{O})) \mapsto \text{false} \} \]
\[ |(M_{\text{ext}}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_t \neq \emptyset \} \]
\[ \Sigma_P \setminus \Sigma_P' = \{(M_{\text{ext}}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{true} \} \]
\[ \cup \{(M_{\text{ext}}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \cdot \text{oneseq}'(\mathcal{I} \cup \mathcal{O})) \mapsto \text{false} \} \]
\[ |(M_{\text{ext}}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_f \neq \emptyset \} \]
where \( \mathcal{I} = \mathcal{I}', \mathcal{O} = \mathcal{O}' \) and \( M_{\text{ext}} = M_{\text{ext}'} \) |
| 5    | different output that is functionally equivalent to old output:  
\[ (\Sigma_P \neq \Sigma_P') \land (\Sigma_P \equiv \Sigma_P') \]
where \( \mathcal{I} = \mathcal{I}', (\mathcal{O} \neq \mathcal{O}') \land (\mathcal{O} \equiv \mathcal{O}') \) and \( M_{\text{ext}} = M_{\text{ext}'} \) |
| 6    | enforcing restrictions on program state:  
\[ \Sigma_P' \setminus \Sigma_P = \{(M_{\text{ext}}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{false} \} \]
\[ |(M_{\text{ext}}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_{\text{arbi}} \neq \emptyset \} \]
\[ \Sigma_P \setminus \Sigma_P' = \{(M_{\text{ext}}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{true} \} \]
\[ |(M_{\text{ext}}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_{\text{arbi}} \neq \emptyset \} \]
where \( \mathcal{I} = \mathcal{I}', \mathcal{O} = \mathcal{O}' \) and \( M_{\text{ext}} = M_{\text{ext}'} \) |

Figure 3.2: Six Cases of Formalized General New Program Behavior - Part2
to “hello”. Case 6 is backward compatible in that the new program makes implicit specification of the program explicit by enforcing restrictions on program state and therefore eliminating undesired I/O sequence.

The six cases in Fig. 3.1 and 3.2 have covered the changes of output, including more or less or different output. There exists more specific cases of backward compatible program behavior changes under various specific assumptions. However, these more specific cases could be attributed to one of the six cases as far as the changes of output are concerned. In conclusion, it is not possible to go much beyond the six cases of backward compatibility in Fig. 3.1 and 3.2.

We say that a program includes backward compatible updates (against another version of the program) if the updates in the program leads to backward compatible program behavior in DSU (from the other version of the program).
Chapter 4

A FRAMEWORK OF SEMANTIC EQUIVALENCE FOR INTERACTIVE PROGRAMS

We first briefly introduce the formal language that is used to describe the formal program in our study. Then we present our framework of program equivalence. The framework of equivalence facilitates our proof of backward compatibility for real world program changes.

4.1 Formal Programming Language

We present the formal programming language based on which we prove our semantic equivalence results which are used to describe categories of backward compatible changes in Chapter 5.1. We first explain the language syntax, then the language semantics.

4.1.1 Language Syntax

We design our formal language with reference to a number of work [49, 93, 4, 83, 41, 43, 40, 25, 34, 62, 22]. The language syntax is in Figure 4.1. We use \( id \) to range over the set of identifiers, \( n \) to range over integers, \( l \) to range over labels. We assume unique identifiers across all syntactic categories, unique labels across all enumeration types and the prompt type. We have base type Int and Long for integer values. The integers defined in type Int are also defined in type Long. Every label defined in the prompt type is related with an integer constant as the actual value used in output statement. We differentiate type Long and Int to define the bug fix of type relaxation from Int to Long to prevent overflow in calculation (e.g., \( a + b \) can cause an error.
Identifer id  Constant n  Label l

Enum Items \( el ::= l | el_1, el_2 \)

Enumeration \( EN ::= \emptyset | \text{enum } \{ el \} | EN_1, EN_2 \)

Prompt Msg \( msg ::= l: n | msg_1, msg_2 \)

Prompts \( Pmpt ::= \emptyset | \{ msg \} \)

Base Type \( \tau ::= \text{Int} | \text{Long} | \text{pmpt} | \text{enum } id \)

Variables \( V ::= \emptyset | \tau id | \tau id[n] | V_1, V_2 \)

Left Value \( lval ::= id | id_1[id_2] | id[n] \)

Expression \( e ::= id == l | lval | \text{other} \)

Statement \( s ::= lval := e | \text{input } id | \text{output } e | \text{skip} \)

| Stmt Seq. \( S ::= s_1; \ldots; s_k \) for \( k \geq 1 \)

Program \( P ::= Pmpt; EN; V; S_{entry} \)

---

Figure 4.1: Abstract Syntax

with Int but not with Long). The type Int is necessary reflecting the concern of space and time efficiency in practical computation. We also have user-defined enumeration type, prompt type and array type.

We explicitly have “\( id == l \)” and \( lval \) as expressions for convenience of the definition of specific updates. To make our programming language general and to separate the concern of expression evaluation, we parameterize the language by “other” expressions which are unspecified.

We have explicit input and output statement because we model the program behavior as the I/O sequence which is the observational behavior of a program. The I/O statement makes it convenient for the argument of program behavior correspondence. In this dissertation, every I/O value is an integer value which is a common I/O representation [41]. A Statement sequence is defined as \( s_1; \ldots; s_k \) where \( k > 0 \) for the convenience of syntax-direct definition from both ends of the sequence.
### Values

<table>
<thead>
<tr>
<th>Values</th>
<th>$v \in \mathbb{Z}_L \cup \mathbb{L}$</th>
<th>integer values in type long and enum/prompt labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O Values</td>
<td>$v_{io} \in \mathbb{Z}_L$</td>
<td>tagged input values</td>
</tr>
<tr>
<td>Inputs</td>
<td>$v_i ::= v_{io}$</td>
<td>tagged input values</td>
</tr>
<tr>
<td>Eval. Values</td>
<td>$v_{err} ::= v</td>
<td>\text{error}$</td>
</tr>
<tr>
<td>Param. Types</td>
<td>$\tau ::= \tau</td>
<td>\text{array}(\tau, n)$</td>
</tr>
<tr>
<td>Loop Labels</td>
<td>$\text{loop}_{\beta} \in \mathbb{N}$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2: Values, Types and Domains

A program is composed of a possibly empty prompt type $Pmpt$, a possibly empty sequence of enumeration types $EN$, a possibly empty sequence of global variables $V$ and a sequence of entry statements $S_{\text{entry}}$. Finally, we have a standard type system based on our syntax.

#### 4.1.2 Small-Step Operational Semantics

Figure 4.2 shows semantic categories of our language. We consider values to be either labels $\mathbb{L}$ or integer numbers $\mathbb{Z}_L$ defined in type Long. The integer numbers defined $\mathbb{Z}_I$ of type Int are a proper subset of those in type Long, $\mathbb{Z}_I \subset \mathbb{Z}_L$. We use the notation $\mathbb{Z}_L+$ for the positive integers defined in type Long. We use the notation $\text{udf}[\tau]$ for an undefined value of type $\tau$. Unlike the “undefined” in Clight [22], we need to parameterize the undefined value with a type $\tau$ because we do not have an underlining memory model that can interpret any block content according to a type. An individual value in I/O sequence is an integer number with tag differentiating inputs and outputs, our tags for inputs and outputs are standard notations [41]. The value from expression evaluation is a pair. One of the pair is either a value $v$ or
Figure 4.3: Elements of an Execution State

“error” for runtime errors (e.g., division by zero); the other is the overflow flag (i.e., 0 for no overflow).

We use notation $\tau^\top$ for all types that are defined in syntax, including array types.

Every loop statement in a program is with a unique label $\text{loop}_{\text{lbl}}$ of a natural number in order to differentiate their executions.

The composition of an execution state is in Figure 4.3.

1. The crash flag $f$ is initially zero and is set to one whenever an exception occurs. Once the crash flag is set, it is not cleared. We only consider unrecoverable crashes. The crash flag is used to make sure that updates do not occur in error states.

2. The overflow flag $of$ is initially zero and is set to one whenever an integer overflow in expression evaluation occurs. Overflow flag is sticky in the sense that once it is set, the flag is not cleared. According to [32], integer overflows are common in mature programs.
\[(S, m) \rightarrow (S', m') \quad (r, m) \rightarrow (r', m') \quad (E[r], m) \rightarrow (E[r'], m')\]

**Eval. Context**

\[E ::= \cdot | id[E] | E == l | id := E | id[E] := e | id[v] := E | \text{output } E | \text{while } (E) \{S\} | \text{If } (E) \text{ then } \{S_t\} \text{ else } \{S_f\} | E; S\]

Figure 4.4: Contextual Semantic Rule

3. \(\Gamma\) is the type environment mapping enumeration type identifiers and variable identifiers to their types. Type environment is necessary for checking array index out of bound or checking value mismatch in execution of input/assignment statement.

4. Loop counters \(\text{loop}_c\) are to record the number of iterations for one instance of a loop statement. The loop counters \(\text{loop}_c\) is not necessary for program executions but are needed for our reasoning of the execution of loops. When a counter entry for loop label \(n\) is not defined in loop counters \(\text{loop}_c\), we write \(\text{loop}_c(n) = \bot\). Otherwise, we write \(\text{loop}_c(n) \neq \bot\).

5. The value store \(\sigma\) is a valuation for scalar variables, array elements, the input sequence variable, and the I/O sequence variable.

Execution state \(m\) is a composition of elements discussed above. In our SOS rules, we only show components of a state \(m\) when necessary (e.g., \(m(\Gamma, \sigma)\)).

Figure 4.4 shows typical contextual rule and Figure 4.5, 4.6 and 4.7 show all SOS rules.

Figure 4.5 shows rules for expression evaluation. We use the expression meaning function \(E : \text{other} \rightarrow \sigma \rightarrow (v_{err} \times \{0, 1\})\) to evaluate “other” expressions. In evaluation
Figure 4.5: SOS Rules for Expressions

of expression “other” against a value store $\sigma$, the expression meaning function $E$ returns a pair $(v_{err}, of)$ where the value $v_{err}$ is either a value $v$ or an “error”, $of$ is a flag indicating if there is integer overflow in the evaluation (e.g., 1 if there is overflow).

The meaning function $E$ interprets “other” expressions deterministically. In addition, there is a function $Use : \text{other} \rightarrow \{id\}$ maps an “other” expression to a set of variables
\[(r, m) \rightarrow (r', m')\]

\[\text{As-Scl} \quad f = 0 \quad \sigma(id) \neq \bot \quad (id := v, m(f, \sigma)) \rightarrow (\text{skip}, m(\sigma[v/id]))\]

\[\text{As-Arr} \quad f = 0 \quad \sigma(id, v_1) \neq \bot \quad (id[v_1] := v_2, m(f, \sigma)) \rightarrow (\text{skip}, m(\sigma[v_2/(id, v_1)]))\]

\[\text{As-Err1} \quad f = 0 \quad (\Gamma \vdash id : \text{array}(r, n)) \land (1 \leq v_1 \leq n) \quad (id[v_1] := v_2, m(f, \Gamma')) \rightarrow (id[v_1] := v_2, m(1/f))\]

\[\text{As-Err2} \quad f = 0 \quad \sigma(id) \neq \bot \quad (\Gamma \vdash id : \text{Int}) \land (v \in (\mathbb{Z}_L \setminus \mathbb{Z}_I)) \quad (id := v, m(f, \Gamma, \sigma)) \rightarrow (id := v, m(1/f))\]

\[\text{As-Err3} \quad f = 0 \quad \sigma(id, v_1) \neq \bot \quad (\Gamma \vdash id : \text{array}(\text{Int}, n)) \land (v_2 \in (\mathbb{Z}_L \setminus \mathbb{Z}_I)) \quad (id[v_1] := v_2, m(f, \Gamma, \sigma)) \rightarrow (id[v_1] := v_2, m(1/f))\]

\[\text{If-T} \quad f = 0 \quad (v \in \mathbb{Z}_L) \land (v \neq 0) \quad (\text{If } v \text{ then } \{S_i\} \text{ else } \{S_f\}, m(f)) \rightarrow (S_i, m)\]

\[\text{If-F} \quad f = 0 \quad (\text{If } 0 \text{ then } \{S_i\} \text{ else } \{S_f\}, m(f)) \rightarrow (S_f, m)\]

\[\text{Wh-T} \quad f = 0 \quad (v \in \mathbb{Z}_L) \land (v \neq 0) \quad \text{loop}_c(n) = k \quad (\text{while}_{(n)} (v) \{S\}, m(f, \text{loop}_c)) \rightarrow (S; \text{while}_{(n)} (e) \{S\}, m(\text{loop}_c[(k + 1)/n]))\]

\[\text{Wh-F} \quad f = 0 \quad \text{loop}_c(n) \neq \bot \quad (\text{while}_{(n)} (0) \{S\}, m(f, \text{loop}_c)) \rightarrow (\text{skip}, m(\text{loop}_c[0/n]))\]

\[\text{Seq} \quad f = 0 \quad (\text{skip}, S, m(f)) \rightarrow (S, m)\]

\[\text{Crash} \quad f = 1 \quad (s, m(f)) \rightarrow (s, m)\]

Figure 4.6: SOS Rules for Assignment, If, and While Statements

used in the expression; there is a function Err : other → \{id\} maps an expression to a set of variables whose values decide if the evaluation of expression leads to crash. We
\[ (r, m) \rightarrow (r', m') \]

Figure 4.7: SOS Rules for Input/Output Statements

assume function Use and Err available. The value returned by the expression meaning
function only depends on the values of variables in the use set of the expression and the error evaluation only depends on the variables in the error set.

As to integer overflow, there are two ways of handling overflow in practice one is to wrap around overflow using two-complement representation (e.g., the gcc option -fwrapv); the other is to generates traps for overflow (e.g., the gcc option -ftrapv). We adopt a combination of the two handling of overflow: the meaning function $E$ wraps the overflow in some representation (e.g., two-complement) and notifies the overflow in return value. Rule EOflow-1 and EOflow-2 update the sticky overflow flag. The evaluation of $lval$ or $id == l$ is shown by respective rules in Figure 4.5.

Figure 4.6 shows SOS rules for assignment, If, while statements, statement sequence, and crash, which are almost standard. There are four particular crashes in execution of assignment statements. One is array out of bound for array access for l-value (e.g., rule As-Err1); the second is assigning a value defined in type Long but not type Int to an Int-typed variable (e.g., rule As-Err2); the third is value mismatch in input statement; the last is expression evaluation exception. As to loop statement, if the predicate expression evaluates to a nonzero integer, corresponding loop counter value increments by one; otherwise, the loop counter value is reset to zero. We use rule Crash to treat crash as non-terminating execution, telling apart normally terminating executions and others.

Figure 4.7 shows rules for the execution of input/output statements. As to input, there is conversion from values of type Long to those of Int or enumeration types but not the prompt type. For an enumeration type, the Long-typed value is transformed to the label with index of that value if possible. There is crash when value conversion is impossible. Besides, there is crash when executing input statement with empty input sequence. We use standard list operation hd and tl for fetching the list head (leftmost element) or the list tail (the list by removing its head) respectively [83].
Last, we construct initial state in following steps: First, crash flag $f$, overflow flag $o$ are zero. Second, type environment is obtained after parsing of the program. Third, every loop counter value in loop $c$ is initially zero. Fourth, every scalar variable or array element has an entry in value store with some initial value if specified. Last, there is initial input sequence and empty I/O sequence.

4.2 Execution

We consider several types of program changes that are allowed by “observational equivalence” without user assumptions. These changes include: statement reordering or duplication, extra statements unrelated to output (e.g., logging related changes), loop fission or fusion, and extra statements unrelated to output. Our program equivalence ensures two programs produce the same output, which means two programs produce same I/O sequence till any output. The program equivalence is established upon two other kinds of equivalence, namely equivalent terminating computation of a variable and equivalent termination behavior.

We first define terminating and non-terminating execution. Then we present the framework of program equivalence in three steps in which every later step relies on prior ones. We first propose a proof rule ensuring two programs to compute a variable in the same way. We then suggest a condition ensuring two programs to either both terminate or both do not terminate. Finally we describe a condition ensuring two programs to produce the same output sequence. Our proof rule of program equivalence gives program point mapping as well as program state mapping. Though we express the program equivalence as a whole program relation, it is easy to apply the equivalence check for local changes using our framework under user’s various assumptions for equivalence.
4.2.1 Terminating Execution and Non-terminating Execution

We define an execution to be a sequence of configurations which are pairs \((S, m)\) where \(S\) is a statement sequence and \(m\) is an execution state shown in Figure 4.3. Let \((S_1, m_1)\), \((S_2, m_2)\) be two consecutive configurations in an execution, the later configuration \((S_2, m_2)\) is obtained by applying one semantic rule w.r.t to the configuration \((S_1, m_1)\), denoted \((S_1, m_1) \rightarrow (S_2, m_2)\), called one step (of execution). For our convenience, we use the notation \((S, m) \rightarrow^k (S', m')\) for \(k > 0\). When we do not care the exact (finite) number of steps, we write the execution as \((S, m) \rightarrow^* (S', m')\). We express terminating executions, non-terminating executions including crash in Definition 7 and 8.

**Definition 7. (Terminating Execution)** A statement sequence \(S\) normally terminates when started in a state \(m\) iff \((S, m) \rightarrow^* (\text{skip}, m'(f))\) where \(f = 0\).

**Definition 8. (Non-terminating Execution)** A statement sequence \(S\) does not terminate when started in a state \(m\) iff, \(\forall k > 0 : (S, m) \rightarrow^k (S_k, m_k)\) where \(S_k \neq \text{skip}\).

4.3 Equivalent Computation for Terminating Programs

We propose a proof rule under which two terminating programs are computing a variable in the same way. We start by giving the definition of equivalent computation for terminating programs right after this paragraph. Then we present the proof rule of equivalent computation in the same way. We prove that the proof rule ensures equivalent computation for terminating programs by induction on the program size of the two programs in the proof rule. We also list auxiliary lemmas required by the soundness proof for the proof rule for equivalent computation for terminating programs.
Definition 9. (Equivalent Computation for Terminating Programs) Two statement sequences $S_1$ and $S_2$ compute a variable $x$ equivalently when started in states $m_1$ and $m_2$ respectively, written $(S_1, m_1) \equiv_x (S_2, m_2)$, iff $(S_1, m_1) \rightarrow^* (\text{skip}, m'_1(\sigma'_1))$ and $(S_2, m_2) \rightarrow^* (\text{skip}, m'_2(\sigma'_2))$ imply $\sigma'_1(x) = \sigma'_2(x)$.

4.3.1 Proof Rule for Equivalent Computation for Terminating Programs

We define a proof rule under which $(S_1, m_1) \equiv_x (S_2, m_2)$ holds for generally constructed initial states $m_1$ and $m_2$, written $S_1 \equiv^S_x S_2$. Our proof rule for equivalent computation for terminating programs allows updates including statement reordering or duplication, loop fission or fusion, additional statements unrelated to the computation and statements movement across if-branch.

Definition 12 includes the recursive proof rule of equivalent computing for terminating programs. The base case is the condition for two simple statements in Definition 11. Definition 10 of imported variables captures the variable def-use chain which is the essence of our equivalence. In Definition 10, the Def and Use refer to variables defined or used in a statement (sequence) or an expression similar to those in the optimization chapter in the dragon book [13]; $S^i$ refers to $i$ consecutive copies of a statement sequence $S$.

Definition 10. (Imported Variables) The imported variables in a sequence of statements $S$ relative to variables $X$, written $\text{Imp}(S, X)$, are defined in one of the following cases:

1. $\text{Def}(S) \cap X = \emptyset$: $\text{Imp}(S, X) = X$;

2. $S = \text{"id := e" or \"input id" or \"output e" and } \text{Def}(S) \cap X \neq \emptyset$:

$$\text{Imp}(S, X) = \text{Use}(S) \cup (X \setminus \text{Def}(S));$$
3. $S = "\text{If } (e) \text{ then } \{S_t\} \text{ else } \{S_f\}"$ and $\text{Def}(S) \cap X \neq \emptyset$:

$$\text{Imp}(S, X) = \text{Use}(e) \cup \bigcup_{y \in X} \left( \text{Imp}(S_t, \{y\}) \cup \text{Imp}(S_f, \{y\}) \right);$$

4. $S = \text{"while}(e) \{S'\}"$ where $(\text{Def}(S') \cap X) \neq \emptyset$:

$$\text{Imp}(S, X) = \bigcup_{i \geq 0} \text{Imp}(S'^i, \text{Use}(e) \cup X);$$

5. For $k > 0$, $S = s_1; \ldots; s_{k+1}$:

$$\text{Imp}(S, X) = \text{Imp}(s_1; \ldots; s_k, \text{Imp}(s_{k+1}, X))$$

**Definition 11. (Base Cases of the Proof Rule for Equivalent Computation for Terminating Programs)** Two simple statements $s_1$ and $s_2$ satisfy the proof rule of equivalent computation of a variable $x$, written $s_1 \equiv^S_x s_2$, iff one of the following holds:

1. $s_1 = s_2$;

2. $s_1 \neq s_2$ and one of the following holds:

   (a) $s_1 = \text{"input id}_1\"$, $s_2 = \text{"input id}_2\", x \notin \{\text{id}_1, \text{id}_2\}$;

   (b) Case a) does not hold and $x \notin \text{Def}(s_1) \cup \text{Def}(s_2)$;

**Definition 12. (Proof Rule of Equivalent Computation for Terminating Programs)** Two statement sequences $S_1$ and $S_2$ satisfy the proof rule of equivalent computation of a variable $x$, written $S_1 \equiv^S_x S_2$, iff one of the following holds:

1. $S_1$ and $S_2$ are one statement and one of the following holds:

   (a) $S_1$ and $S_2$ are simple statement: $s_1 \equiv^S_x s_2$;

   (b) $S_1 = \text{"If } (e) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}$, $S_2 = \text{"If } (e) \text{ then } \{S_2^t\} \text{ else } \{S_2^f\}$" such that all of the following hold:
• \( x \in \text{Def}(S_1) \cap \text{Def}(S_2) \);  
• \( (S'_1 \equiv^x S'_2) \land (S'_f \equiv^x S'_2) \);  

(c) \( S_1 = "\text{while}_{(n_1)}(e) \{ S'_1 \}\)  
\( S_2 = "\text{while}_{(n_2)}(e) \{ S'_2 \}\) such that both of the following hold:  
• \( x \in \text{Def}(S_1) \cap \text{Def}(S_2) \);  
• \( \forall y \in \text{Imp}(S_1, \{ x \}) \cup \text{Imp}(S_2, \{ x \}) : S''_1 \equiv^y S''_2 \);  

(d) \( S_1 \) and \( S_2 \) do not define the variable \( x \): \( x \notin \text{Def}(S_1) \cup \text{Def}(S_2) \).  

2. \( S_1 \) and \( S_2 \) are not both one statement and one of the following holds:  
(a) \( S_1 = S'_1; s_1, S_2 = S'_2; s_2 \) and last statements both define the variable \( x \) such that both of the following hold:  
• \( \forall y \in \text{Imp}(s_1, \{ x \}) \cup \text{Imp}(s_2, \{ x \}) : S'_1 \equiv^y S'_2 \);  
• \( s_1 \equiv^x s_2 \) where \( x \in \text{Def}(s_1) \cap \text{Def}(s_2) \);  

(b) Last statement in \( S_1 \) or \( S_2 \) does not define the variable \( x \):  
\( (x \notin \text{Def}(s_2) \land (S_1 \equiv^x S'_2)) \lor (x \notin \text{Def}(s_1) \land (S'_1 \equiv^x S_2)) \);  

(c) \( S_1 = S'_1; s_1, S_2 = S'_2; s_2 \) and there are statements moving in/out of If statement: \( s_1 = "\text{If}(e) \text{then} \{ S'_1 \} \text{else} \{ S'_f \}\)  
\( s_2 = "\text{If}(e) \text{then} \{ S'_2 \} \text{else} \{ S'_f \}\) such that none of the above cases hold and all of the following hold:  
• \( \forall y \in \text{Use}(e) : S'_1 \equiv^y S'_2 \);  
• \( (S'_1; S'_1 \equiv^x S'_2; S'_2) \land (S'_f; S'_f \equiv^x S'_2; S'_2) \);  

The generalization of definition \( S_1 \equiv^S_x S_2 \) to a set of variables is as follows.  

**Definition 13.** Two statement sequences \( S_1 \) and \( S_2 \) have equivalent computation of variables \( X \), written \( S_1 \equiv^S_X S_2 \), iff \( \forall x \in X : S_1 \equiv^S_x S_2 \).
4.3.2 Soundness of the Proof Rule for Equivalent Computation for Terminating Programs

We show that if two programs satisfy the proof rule of equivalent computation of a variable $x$ (Definition 12) and their value stores in initial states agree on values of the imported variables relative to $x$, then the two programs compute the same value of $x$ if they terminate. We start by proving the theorem for the base cases of terminating computation equivalently.

**Theorem 1.** If $s_1$ and $s_2$ are simple statements that satisfy the proof rule for equivalent computation of $x$, $s_1 \equiv_x s_2$, and their initial states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ agree on the values of the imported variables relative to $x$, $\forall y \in Imp(s_1, \{x\}) \cup Imp(s_2, \{x\}) : \sigma_{s_1}(y) = \sigma_{s_2}(y)$, then $s_1$ and $s_2$ equivalently compute $x$ when started in states $m_1$ and $m_2$ respectively, $(s_1, m_1) \equiv_x (s_2, m_2)$.

The proof is a case analysis according to the cases in the definition of the proof rule for equivalent computation (i.e., Definition 11). Refer to our technical report for details of the proof [88].

**Theorem 2.** If statement sequence $S_1$ and $S_2$ satisfy the proof rule of equivalent computation of a variable $x$, $S_1 \equiv_S S_2$, and their initial states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ agree on the initial values of the imported variables relative to $x$, $\forall y \in Imp(S_1, \{x\}) \cup Imp(S_2, \{x\}) : \sigma_1(y) = \sigma_2(y)$, then $S_1$ and $S_2$ equivalently compute the variable $x$ when started in state $m_1$ and $m_2$ respectively, $(S_1, m_1) \equiv_x (S_2, m_2)$.

The proof of Theorem 2 is by induction on the sum of the program size of $S_1$ and $S_2$. Refer to our technical report for details of the proof [88].
4.3.3 Supporting Lemmas for the Soundness Proof of Equivalent Computation for Terminating Programs

The lemmas include the proof of two while statements computing a variable equivalently used in the proof of Theorem 2 and the property that two programs have same imported variables relative to a variable $x$ if the two programs satisfy the proof rule of equivalent computation of the variable $x$. From the proof rule of terminating computation of a variable $x$ equivalently, we have the two programs either both define $x$ or both do not.

**Lemma 4.3.1.** Let $s_1 = \text{"while}_{(m_1)}(e) \{S_1\}$ and $s_2 = \text{"while}_{(m_2)}(e) \{S_2\}$ be two while statements with the same set of imported variables relative to a variable $x$ (defined in $s_1$ and $s_2$), $\text{Imp}(x)$, and whose loop bodies $S_1$ and $S_2$ terminatingly compute the variables in $\text{Imp}(x)$ equivalently when started in states that agree on the values of the variables imported by $S_1$ or $S_2$ relative to $\text{Imp}(x)$:

- $x \in \text{Def}(s_1) \cap \text{Def}(s_2)$;
- $\text{Imp}(s_1, \{x\}) = \text{Imp}(s_2, \{x\}) = \text{Imp}(x)$;
- $\forall y \in \text{Imp}(x), \forall m_{S_1}(\sigma_{S_1}), m_{S_2}(\sigma_{S_2})$:
  
  $((\forall z \in \text{Imp}(S_1, \text{Imp}(x)) \cup \text{Imp}(S_2, \text{Imp}(x)) : \sigma_{S_1}(z) = \sigma_{S_2}(z)) \Rightarrow (S_1, m_{S_1}(\sigma_{S_1})) \equiv_y (S_2, m_{S_2}(\sigma_{S_2})))$.

If the executions of $s_1$ and $s_2$ terminate when started in states $m_1(\text{loop}_c^1, \sigma_1)$ and $m_2(\text{loop}_c^2, \sigma_2)$ in which $s_1$ and $s_2$ have not already executed (loop counter initially 0: $\text{loop}_c^1(n_1) = \text{loop}_c^2(n_2) = 0$), and whose value stores $\sigma_1$ and $\sigma_2$ agree on the values of the variables in $\text{Imp}(x)$, $\forall y \in \text{Imp}(x), \sigma_1(y) = \sigma_2(y)$, then, for any positive integer $i$, one of the following holds:
1. The loop counters for $s_1$ and $s_2$ are always less than $i$:

$\forall m'_1, m'_2$ such that $(s_1, m_1) \xrightarrow{\ast} (S'_1, m'_1(\text{loop}^1_c))$ and $(s_2, m_2) \xrightarrow{\ast} (S'_2, m'_2(\text{loop}^2_c))$,

$\text{loop}^1_c(n_1) < i$ and $\text{loop}^2_c(n_2) < i$;

2. There are two configurations $(s_1, m_1)$ and $(s_2, m_2)$ reachable from $(s_1, m_1)$ and $(s_2, m_2)$, respectively, in which the loop counters of $s_1$ and $s_2$ are equal to $i$ and value stores agree on the values of imported variables relative to $x$ and, for every state in execution, $(s_1, m_1) \xrightarrow{\ast} (s_1, m_1)$ or $(s_2, m_2) \xrightarrow{\ast} (s_2, m_2)$ the loop counters for $s_1$ and $s_2$ are less than or equal to $i$ respectively:

$\exists (s_1, m_1), (s_2, m_2) : (s_1, m_1) \xrightarrow{\ast} (s_1, m_1(\text{loop}^1_c, \sigma_1, i)) \land (s_2, m_2) \xrightarrow{\ast} (s_2, m_2(\text{loop}^2_c, \sigma_2, i))$

where

- $\text{loop}^1_c(n_1) = \text{loop}^2_c(n_2) = i$; and
- $\forall y \in \text{Imp}(x) : \sigma_1(y) = \sigma_2(y)$ and
- $\forall m'_1 : (s_1, m_1) \xrightarrow{\ast} (S'_1, m'_1(\text{loop}^1_c)) \xrightarrow{\ast}
  (s_1, m_1(\text{loop}^1_c, \sigma_1, i)), \text{loop}^1_c(n_1) \leq i$; and
- $\forall m'_2 : (s_2, m_2) \xrightarrow{\ast} (S'_2, m'_2(\text{loop}^2_c)) \xrightarrow{\ast}
  (s_2, m_2(\text{loop}^2_c, \sigma_2, i)), \text{loop}^2_c(n_2) \leq i$;

The proof is by induction on $i$. Refer to our technical report for details [88].

**Lemma 4.3.2.** Let $s_1 = \text{"while}_{(n_1)}(e) \{S_1 \}$” and $s_2 = \text{"while}_{(n_2)}(e)\{S_2 \}$” be two while statements with the same set of imported variables relative to a variable $x$ (defined in $s_1$ and $s_2$), and whose loop bodies $S_1$ and $S_2$ terminally compute the variables in $\text{Imp}(x)$ equivalently when started in states that agree on the values of the variables imported by $S_1$ or $S_2$ relative to $\text{Imp}(x)$:

- $x \in \text{Def}(s_1) \cap \text{Def}(s_2)$;
• \( \text{Imp}(s_1, \{x\}) = \text{Imp}(s_2, \{x\}) = \text{Imp}(x) \):

• \( \forall y \in \text{Imp}(x) \forall m_{S_1}(\sigma_{S_1}) m_{S_2}(\sigma_{S_2}) : \\
(\forall z \in \text{Imp}(S_1, \text{Imp}(x)) \cup \text{Imp}(S_2, \text{Imp}(x)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \Rightarrow ((S_1, m_{S_1}(\sigma_{S_1})) \equiv_y (S_2, m_{S_2}(\sigma_{S_2}))). \)

If the executions of \( s_1 \) and \( s_2 \) terminate when started in states \( m_1(\text{loop}_1^1, \sigma_1) \) and \( m_2(\text{loop}_2^2, \sigma_2) \) in which \( s_1 \) and \( s_2 \) have not already executed (loop counter initially 0: \( \text{loop}_1^1(n_1) = \text{loop}_2^2(n_2) = 0 \)), and whose value stores \( \sigma_1 \) and \( \sigma_2 \) agree on the values of the variables in \( \text{Imp}(x) \), \( \forall y \in \text{Imp}(x) \sigma_1(y) = \sigma_2(y) \), when \( s_1 \) and \( s_2 \) terminate, \( (s_1, m_1) \rightarrow (\text{skip}, m_1, (\sigma'_1)) \) and \( (s_2, m_2) \rightarrow (\text{skip}, m_2, (\sigma'_2)) \), value stores \( \sigma'_1 \) and \( \sigma'_2 \) agree on the value of \( x \), \( \sigma'_1(x) = \sigma'_2(x) \).

We show that there must exist a finite integer \( k \) such that the loop counters of \( s_1 \) and \( s_2 \) in executions started in states \( m_1 \) and \( m_2 \) is always less than \( k \). Refer to our technical report for details of the proof [88].

**Lemma 4.3.3.** If two statement sequences \( S_1 \) and \( S_2 \) satisfy the proof rule of terminating computation of a variable \( x \) equivalently, then \( S_1 \) and \( S_2 \) have same imported variables relative to \( x \): \( (S_1 \equiv^*_2 S_2) \Rightarrow (\text{Imp}(S_1, \{x\}) = \text{Imp}(S_2, \{x\})) \).

By induction on \( \text{size}(S_1) + \text{size}(S_2) \), the sum of the program size of \( S_1 \) and \( S_2 \). Refer to our technical report for details [88].

### 4.4 Termination in the Same Way

We proceed to propose a proof rule under which two statement sequences either both terminate or both do not terminate. We start by giving the definition of termination in the same way. Then we present the proof rule of termination in the same way. Our proof rule of termination in the same way allows updates such as
statement duplication or reordering, loop fission or fusion and additional terminating statements. We prove that the proof rule ensures terminating in the same way by induction on the program size of the two programs in the proof rule. We also list auxiliary lemmas required by the proof of termination in the same way.

**Definition 14. (Termination in the Same Way)** Two statement sequences $S_1$ and $S_2$ terminate in the same way when started in states $m_1$ and $m_2$ respectively, written $(S_1, m_1) \equiv_H (S_2, m_2)$, iff one of the following holds:

1. $(S_1, m_1) \rightarrow^* (\text{skip}, m'_1)$ and $(S_2, m_2) \rightarrow^* (\text{skip}, m'_2)$;

2. $\forall i \geq 0, (S_1, m_1) \rightarrow (S^i_1, m^i_1)$ and $(S_2, m_2) \rightarrow (S^i_2, m^i_2)$ where $S^i_1 \neq \text{skip}, S^i_2 \neq \text{skip}$.

**4.4.1 Proof Rule for Termination in the Same Way**

We define proof rules under which two statement sequences $S_1$ and $S_2$ terminate in the same way. We summarize the cause of non-terminating execution and then give the proof rule.

We consider two causes of nonterminating executions: crash and infinite iterations of loop statements. As to crash [71], we consider four common causes based on our language: expression evaluation exceptions, the lack of input value, input/assignment value type mismatch and array index out of bound. In essence, the causes of nontermination are partly due to the values of some particular variables during executions. We capture variables affecting each source of nontermination; loop deciding variables $\text{LVar}(S)$ are variables affecting the evaluation of a loop statements in the statement sequence $S$, crash deciding variables $\text{CVar}(S)$ are variables whose values decide if a crash occurs in $S$. We list the definitions of $\text{LVar}(S)$ and $\text{CVar}(S)$ in Definition 15.
and 16. Definition 17 summarizes the variables whose values decide if one program terminates.

**Definition 15. (Loop Deciding Variables)** The loop deciding variables of a statement sequence $S$, written $LVar(S)$, are defined as follows:

1. $LVar(S) = \emptyset$ if $\not\exists s = \text{“while(e) \{S’\}”}$ and $s \in S$;

2. $LVar(\text{“If (e) then \{S_t\} else \{S_f\}”}) = Use(e) \cup LVar(S_t) \cup LVar(S_f)$ if “while(e){S’}” $\in S$;

3. $LVar(\text{“while(e){S’}”}) = Imp(S, Use(e) \cup LVar(S'))$;

4. For $k > 0$, $LVar(s_1;...;s_k; s_k+1) = LVar(s_1;...;s_k) \cup Imp(s_1;...;s_k, LVar(s_k+1))$;

**Definition 16. (Crash Deciding Variables)** The crash deciding variables of a statement sequence $S$, written $CVar(S)$, are defined as follows:

1. $CVar(\text{skip}) = \emptyset$;

2. $CVar(lval := e) = Idx(lval) \cup Use(e)$ if $(\Gamma \vdash lval : \text{Int}) \land (\Gamma \vdash e : \text{Long})$;

3. $CVar(lval := e) = Idx(lval) \cup Err(e)$ if $(\Gamma \vdash lval : \text{Int}) \land (\Gamma \vdash e : \text{Long})$ does not hold;

4. $CVar(\text{input id}) = \{id_I\}$;

5. $CVar(\text{output e}) = Err(e)$;

6. $CVar(\text{“If (e) then \{S_t\} else \{S_f\}”}) = Err(e)$, if $CVar(S_t) \cup CVar(S_f) = \emptyset$;

7. $CVar(\text{“If (e) then \{S_t\} else \{S_f\}”}) = Use(e) \cup CVar(S_t) \cup CVar(S_f)$, if $CVar(S_t) \cup CVar(S_f) \neq \emptyset$;
8. $CVar("\text{while}_{(n)}(e)\{S^r\})") = Imp("\text{while}_{(n)}(e)\{S^r\}", Use(e) \cup CVar(S^r));$

9. For $k > 0$, $CVar(s_1;...;s_{k+1}) = CVar(s_1;...;s_k) \cup Imp(s_1;...;s_k, CVar(s_{k+1}));$

**Definition 17. (Termination Deciding Variables)** The termination deciding variables of statement sequence $S$ are $CVar(S) \cup LVar(S)$, written $TVar(S)$.

**Definition 18. (Base Cases of the Proof Rule of Termination in the Same Way)** Two simple statements $s_1$ and $s_2$ satisfy the proof rule of termination in the same way, written $s_1 \equiv_s s_2$, iff one of the following holds:

1. $s_1$ and $s_2$ are same, $s_1 = s_2$;

2. $s_1$ and $s_2$ are input statement with same type variable: $s_1 = \text{"input id}_1\text{"}, s_2 = \text{"input id}_2\text{"}$ where $(\Gamma_{s_1} \vdash id_1 : \tau) \land (\Gamma_{s_2} \vdash id_2 : \tau)$;

3. $s_1 = \text{"output e\" or } id_1 := e\text{"}, s_2 = \text{"output e\" or } id_2 := e\text{"}$ where both of the following hold:
   
   - There is no possible value mismatch in "id$_1$ := e", $\neg(\Gamma_{s_1} \vdash id_1 : \text{Int}) \lor \neg(\Gamma_{s_1} \vdash e : \text{Long}) \lor (\Gamma_{s_1} \vdash e : \text{Int})$.
   - There is no possible value mismatch in "id$_2$ := e", $\neg(\Gamma_{s_2} \vdash id_2 : \text{Int}) \lor \neg(\Gamma_{s_2} \vdash e : \text{Long}) \lor (\Gamma_{s_2} \vdash e : \text{Int})$.

**Definition 19. (Proof Rule of Termination in the Same Way)** Two statement sequences $S_1$ and $S_2$ satisfy the proof rule of termination in the same way, written $S_1 \equiv^S_H S_2$, iff one of the following holds:

1. $S_1$ and $S_2$ are both one statement and one of the following holds.
   
   (a) $S_1$ and $S_2$ are simple statements: $s_1 \equiv^S_H s_2$;
(b) $S_1 = \text{"If}(e) \text{ then } \{S^f_1\} \text{ else } \{S^f_1\}$, $S_2 = \text{"If}(e) \text{ then } \{S^f_2\} \text{ else } \{S^f_2\}$ and one of the following holds:

i. $S^f_1, S^f_1, S^f_2, S^f_2$ are all sequences of “skip”;

ii. At least one of $S^f_1, S^f_1, S^f_2, S^f_2$ is not a sequence of “skip” such that:

$(S^f_1 \equiv^S_H S^f_2) \land (S^f_1 \equiv^S_H S^f_2)$;

(c) $S_1 = \text{"while}_{(n_1)}(e)\{S''_1\}$, $S_2 = \text{"while}_{(n_2)}(e)\{S''_2\}$ and both of the following hold:

- $S''_1 \equiv^S_H S''_2$;
- $S''_1$ and $S''_2$ have equivalent computation of $\text{TVar}(S_1) \cup \text{TVar}(S_2)$;

2. $S_1$ and $S_2$ are not both one statement and one of the following holds:

(a) $S_1 = S'_1; s_1$ and $S_2 = S'_2; s_2$ and all of the following hold:

- $S'_1 \equiv^S_H S'_2$;
- $S'_1$ and $S'_2$ have equivalent computation of $\text{TVar}(s_1) \cup \text{TVar}(s_2)$;
- $s_1 \equiv^S_H s_2$ where $s_1$ and $s_2$ are not “skip”;

(b) One last statement is “skip”:

\[
((S_1 = S'_1; \text{"skip"}) \land (S'_1 \equiv^S_H S'_2)) \lor ((S_2 = S'_2; \text{"skip"}) \land (S_1 \equiv^S_H S'_2)).
\]

(c) One last statement is a “duplicate” statement and one of the following holds:

i. $S_1 = S'_1; s'_1; S''_1; s_1$ and all of the following hold:

- $S'_1; s'_1; S''_1 \equiv^S_H S_2$;
- $(s'_1 \equiv^S_H s_1) \land (s_1 \neq \text{"skip"})$;
- $\text{Def}(s'_1; S''_1) \cap \text{TVar}(s_1) = \emptyset$;

ii. $S_2 = S'_2; s'_2; S''_2; s_2$ and all of the following hold:
4.4.2 Soundness of the Proof Rule for Termination in the Same Way

We show that two statement sequences satisfy the proof rules of termination in the same way, and their initial states agree on the values of their termination deciding variables, then they either both terminate or both do not terminate.

**Theorem 3.** If two simple statements \( s_1 \) and \( s_2 \) satisfy the proof rule of termination in the same way, \( s_1 \equiv^* s_2 \), and their initial states \( m_1(f_1, \sigma_1) \) and \( m_2(f_2, \sigma_2) \) with crash flags not set, \( f_1 = f_2 = 0 \), and whose value stores agree on values of the termination deciding variables of \( s_1 \) and \( s_2 \), \( \forall x \in TVar(s_1) \cup TVar(s_2) : \sigma_1(x) = \sigma_2(x) \), when executions of \( s_1 \) and \( s_2 \) start in states \( m_1 \) and \( m_2 \) respectively, then \( s_1 \) and \( s_2 \) terminate in the same way when started in states \( m_1 \) and \( m_2 \) respectively: \( (s_1, m_1) \equiv_H (s_2, m_2) \).

The proof is a case analysis of those cases in the definition of \( s_1 \equiv^* s_2 \). Refer to our technical report for details of the proof [88].
Theorem 4. If two statement sequences $S_1$ and $S_2$ satisfy the proof rule of termination in the same way, $S_1 \equiv^S_S S_2$, and their respective initial states $m_1(f_1, \sigma_1)$ and $m_2(f_2, \sigma_2)$ with crash flags not set, $f_1 = f_2 = 0$, and whose value stores agree on values of the termination deciding variables of $S_1$ and $S_2$, $\forall x \in TVar(S_1) \cup TVar(S_2) : \sigma_1(x) = \sigma_2(x)$, then $S_1$ and $S_2$ terminate in the same way when started in states $m_1$ and $m_2$ respectively: $(S_1, m_1) \equiv_H (S_2, m_2)$.

The proof is by induction on $\text{size}(S_1) + \text{size}(S_2)$, the sum of program size of $S_1$ and $S_2$. The proof is a case analysis of all cases in the definition of $S_1 \equiv^S_S S_2$. We do induction on loop iterations for the case of two corresponding loop statements. Lemma 4.4.12 in Appendix illustrates how we prove two loop statements to terminate in the same way. Refer to our technical report for details of the proof [88].

4.4.3 Supporting Lemmas for the Soundness Proof of Termination in the Same Way

The supporting lemmas include various properties of $TVar(S)$, two statement sequences satisfying the proof rule of termination in the same way consume the same number of input values when both terminate, and the proof for the case of while statement of theorem 4.

Properties of the Termination Deciding Variables

Lemma 4.4.1. The crash variables of $S_1; S'_1$ is same as the union of the crash variables of $S_1$ and the imported variables in $S_1$ relative to the crash variables of $S'_1$, $CVar(S_1; S'_1) = CVar(S_1) \cup \text{Imp}(S_1, CVar(S'_1))$.

Let $S'_1 = s_1; ...; s_k$ for $k > 0$. We show the lemma by induction on $k$. 

52
Lemma 4.4.2. The loop deciding variables of $S_1; S'_1$ is same as the union of the loop deciding variables of $S_1$ and the imported variables in $S_1$ relative to the loop deciding variables of $S'_1$, $LVar(S_1; S'_1) = LVar(S_1) \cup \text{Imp}(S_1, LVar(S'_1))$.

By proof of Lemma 4.4.2 similar to that of lemma 4.4.1 above.

Lemma 4.4.3. If two statement sequences $S_1$ and $S_2$ satisfy the proof rule of termination in the same way, then $S_1$ and $S_2$ have same loop variables, $(S_1 \equiv^S_H S_2) \Rightarrow (LVar(S_1) = LVar(S_2))$.

By induction on size($S_1$) + size($S_2$), the sum of the program size of $S_1$ and $S_2$.

Lemma 4.4.4. If two statement sequences $S_1$ and $S_2$ satisfy the proof rule of termination in the same way, then $S_1$ and $S_2$ have same crash variables, $(S_1 \equiv^S_H S_2) \Rightarrow (CVar(S_1) = CVar(S_2))$.

By proof similar to those for Lemma 4.4.3.

Corollary 4.4.1. If two statement sequences $S_1$ and $S_2$ satisfy the proof rule of termination in the same way, then $S_1$ and $S_2$ have same termination deciding variables, $(S_1 \equiv^S_H S_2) \Rightarrow (TVar(S_1) = TVar(S_2))$.

By Lemma 4.4.3, and 4.4.4.

Properties of the Input Sequence Variable

Lemma 4.4.5. If there is no input statement in a statement sequence $S$, then the input sequence variable is not in the defined variables of $S$, $(\exists \text{"input } x\text{" } \in S) \Rightarrow id_I \notin \text{Def}(S)$.

Proof. By induction on abstract syntax of $S$. \qed
Lemma 4.4.6. If there is no input statement in a statement sequence $S$, then the input sequence variable is not in the crash variables of $S$, ($\nexists \text{“input } x \text{” } \in S \Rightarrow (id_I \notin CVar(S))$).

Proof. By induction on abstract syntax of $S$.

Lemma 4.4.7. If there is no input statement in a statement sequence $S$, then the input sequence variable is in the loop variables of $S$, ($\nexists \text{“input } x \text{” } \in S \Rightarrow (id_I \notin LVar(S))$).

Proof. By induction on abstract syntax of $S$.

Corollary 4.4.2. If there is no input statement in a statement sequence $S$, then the input sequence variable is in the termination deciding variables of $S$, ($\nexists \text{“input } x \text{” } \in S \Rightarrow (id_I \notin TVar(S))$).

By Lemma 4.4.6 and 4.4.7.

Lemma 4.4.8. If there is one input statement in a statement sequence $S$, then the input sequence variable is in the crash variables and defined variables of $S$, ($\exists \text{“input } x \text{” } \in S \Rightarrow (id_I \in CVar(S)) \wedge (id_I \in Def(S))$).

Proof. By induction on abstract syntax of $S$.

Lemma 4.4.9. If there is one input statement in a statement sequence $S$, then the imported variables in $S$ relative to the input sequence variable are a subset of the crash variables of $S$, ($\exists \text{“input } x \text{” } \in S \Rightarrow (Imp(S, \{id_I\}) \subseteq CVar(S))$).

Proof. By induction on abstract syntax of $S$.

Lemma 4.4.10. If two programs $S_1$ and $S_2$ satisfy the proof rule of termination in the same way, then $S_1$ and $S_2$ satisfy the proof rule of terminating computation in the same way of the input sequence, ($S_1 \equiv^{S}_{H} S_2 \Rightarrow (S_1 \equiv^{S}_{id_{I}} S_2)$).
Proof. By induction on size($S_1$) + size($S_2$).

Lemma 4.4.11. If two programs $S_1$ and $S_2$ satisfy the proof rule of termination in the same way, and $S_1$ and $S_2$ both terminate when started in their initial states with crash flags not set, $f_1 = f_2 = 0$, whose value stores agree on values of variables of the termination deciding variables of $S_1$ and $S_2$, $\forall x \in TVar(S_1) \cup TVar(S_2)$, $\sigma_1(x) = \sigma_2(x)$, and $S_1$ and $S_2$ are fed with the same infinite input sequence, $\sigma_1(id_I) = \sigma_2(id_I)$, $(S_1, m_1(f_1, \sigma_1) ) \rightarrow (skip, m'_1(\sigma'_1))$ and $(S_2, m_2(f_2, \sigma_2) ) \rightarrow (skip, m'_2(\sigma'_2))$, then the execution of $S_1$ and $S_2$ consume the same number of input values, $\sigma'_1(id_I) = \sigma'_2(id_I)$.

Proof. By Lemma 4.4.10, $S_1 \equiv_{id_I} S_2$. By Lemma 4.4.9, $\text{Imp}(S_1, id_I) \subseteq CVar(S_1)$ and $\text{Imp}(S_2, id_I) \subseteq CVar(S_2)$. By assumption, $\forall x \in \text{Imp}(S_1, id_I) \cup \text{Imp}(S_2, id_I) : \sigma_1(x) = \sigma_2(x)$. By Theorem 2, $\sigma'_1(id_I) = \sigma'_2(id_I)$.

Theorem of Two Loop Statements Terminating in the Same Way

Lemma 4.4.12. Let $s_1 = \text{while}_{(n_1)}(e)\{S_1\}$ and $s_2 = \text{while}_{(n_2)}(e)\{S_2\}$ be two while statements with the same set of termination deciding variables in program $P_1$ and $P_2$ respectively, whose bodies $S_1$ and $S_2$ satisfy the proof rule of equivalently computation of variables in $TVar(s)$, and $S_1$ and $S_2$ terminate in the same way when started in states with crash flags not set and agreeing on values of variables in $TVar(S_1) \cup TVar(S_2)$:

- $TVar(s_1) = TVar(s_2) = TVar(s)$;
- $\forall x \in TVar(s) : S_1 \equiv_x S_2$;
- $\forall m_{S_1}(f_{S_1}, \sigma_{S_1}) m_{S_2}(f_{S_2}, \sigma_{S_2})$ :
  $(((\forall z \in TVar(S_1) \cup TVar(S_2)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \land (f_{S_1} = f_{S_2} = 0))$ \Rightarrow $(S_1, m_{S_1}(f_{S_1}, \sigma_{S_1})) \equiv_H (S_2, m_{S_2}(f_{S_2}, \sigma_{S_2}))$. 

55
If \( s_1 \) and \( s_2 \) start in the state \( m_1(f_1, \text{loop}_c^1, \sigma_1) \) and 
\( m_2(f_2, \text{loop}_c^2, \sigma_2) \) respectively in which crash flags are not set, \( f_1 = f_2 = 0, s_1 \) and \( s_2 \) have not already executed, \( \text{loop}_c^1(n_1) = \text{loop}_c^2(n_2) = 0 \), value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of variables in \( \text{TVar}(s) \), \( \forall x \in \text{TVar}(s) : \sigma_1(x) = \sigma_2(x) \), then, for any positive integer \( i \), one of the following holds:

1. The loop counters for \( s_1 \) and \( s_2 \) are less than \( i \) where \( s_1 \) and \( s_2 \) terminate in the same way:
   \[
   \forall m_1' m_2' : (s_1, m_1) \xrightarrow{*} (S_1', m_1' (\text{loop}_c^{1'})) \text{ and } (s_2, m_2) \xrightarrow{*} (S_2', m_2' (\text{loop}_c^{2'})),
   \]
   \( \text{loop}_c^{1'}(n_1) < i \) and \( \text{loop}_c^{2'}(n_2) < i \) and one of the following holds:
   
   (a) \( s_1 \) and \( s_2 \) both terminate:
   \[
   (s_1, m_1) \xrightarrow{*} (\text{skip}, m_1'') \text{ and } (s_2, m_2) \xrightarrow{*} (\text{skip}, m_2'').
   \]
   
   (b) \( s_1 \) and \( s_2 \) both do not terminate:
   \[
   \forall k > 0 : (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k}) \text{ and } (s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k}) \text{ in which } S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}.
   \]

2. The loop counters for \( s_1 \) and \( s_2 \) are less than or equal to \( i \) where \( s_1 \) and \( s_2 \) do not terminate such that there are no configurations \( (s_1, m_{1_i}) \) and \( (s_2, m_{2_i}) \) reachable from \( (s_1, m_1) \) and \( (s_2, m_2) \), respectively, in which crash flags are not set, the loop counters of \( s_1 \) and \( s_2 \) are equal to \( i \), and value stores agree on the values of variables in \( \text{TVar}(s) \):
   
   \[
   \forall m_1' m_2' : (s_1, m_1) \xrightarrow{*} (S_1', m_1' (\text{loop}_c^{1'})), (s_2, m_2) \xrightarrow{*} (S_2', m_2' (\text{loop}_c^{2'})) \text{ where } \text{loop}_c^{1'}(n_1) \leq i, \text{loop}_c^{2'}(n_2) \leq i;
   \]
   
   \[
   \forall k > 0 : (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k}), (s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k}) \text{ where }
   \]

56
$S_{1_i} \neq \text{skip}, S_{2_i} \neq \text{skip};$ and

- $\exists (s_1, m_1), (s_2, m_2):$
  - $(s_1, m_1) \xrightarrow{\ast} (s_1, m_1, (f_1, \text{loop}_{c_i}^1, \sigma_1)) \land$
  - $(s_2, m_2) \xrightarrow{\ast} (s_2, m_2, (f_2, \text{loop}_{c_i}^2, \sigma_2))$ where
    - $f_1 = f_2 = 0;$ and
    - $\text{loop}_{c_i}^1(n_1) = \text{loop}_{c_i}^2(n_2) = i;$ and
    - $\forall x \in \text{TVar}(s): \sigma_{1_i}(x) = \sigma_{2_i}(x).$

3. There are two configurations $(s_1, m_1_i)$ and $(s_2, m_2_i)$ reachable from $(s_1, m_1)$ and $(s_2, m_2)$, respectively, in which both crash flags are not set, the loop counters of $s_1$ and $s_2$ are equal to $i$ and value stores agree on the values of variables in $\text{TVar}(s)$, and for every state in execution $(s_1, m_1) \xrightarrow{\ast} (s_1, m_1_i)$ or $(s_2, m_2) \xrightarrow{\ast} (s_2, m_2_i)$, the loop counters for $s_1$ and $s_2$ are less than or equal to $i$ respectively:

$$\exists (s_1, m_1_i), (s_2, m_2_i): (s_1, m_1) \xrightarrow{\ast} (s_1, m_1_i, (f_1, \text{loop}_{c_i}^1, \sigma_1)) \land (s_2, m_2) \xrightarrow{\ast} (s_2, m_2_i, (f_2, \text{loop}_{c_i}^2, \sigma_2))$$

- $f_1 = f_2 = 0;$ and
- $\text{loop}_{c_i}^1(n_1) = \text{loop}_{c_i}^2(n_2) = i;$ and
- $\forall x \in \text{TVar}(s): \sigma_{1_i}(x) = \sigma_{2_i}(x);$
- $\forall m_1': (s_1, m_1) \xrightarrow{\ast} (S_1', m_1'(m_1')) \xrightarrow{\ast} (s_1, m_1_i),$
  - $\text{loop}_{c_i}^1'(n_1) \leq i;$ and
- $\forall m_2': (s_2, m_2) \xrightarrow{\ast} (S_2', m_2'(m_2')) \xrightarrow{\ast} (s_2, m_2_i),$
  - $\text{loop}_{c_i}^2'(n_2) \leq i;$

The proof is by induction on $i$. Refer to our technical report for details [88].
Corollary 4.4.3. Let \( s_1 = \"while_{(n_1)}(e)\{S_1\}\) and \\
\( s_2 = \"while_{(n_2)}(e)\{S_2\}\) be two while statements respectively, with the same set of 
the termination deciding variables, \( TVar(s_1) = TVar(s_2) = TVar(s) \), whose bodies 
\( S_1 \) and \( S_2 \) satisfy the proof rule of equivalently computation of variables in \( TVar(s) \), 
\( \forall x \in TVar(s) : (S_1) \equiv^S_x (S_2) \), and whose bodies \( S_1 \) and \( S_2 \) terminate in the same way 
when started in states with crash flags not set and agreeing on values of variables in 
\( TVar(S_1) \cup TVar(S_2) \):

\[
\forall m_{S_1}(f_{S_1}, \sigma_{S_1}), m_{S_2}(f_{S_2}, \sigma_{S_2}) : \\
((\forall z \in TVar(S_1) \cup TVar(S_2)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \land (f_{S_1} = f_{S_2} = 0)) \Rightarrow (S_1, m_{S_1}(f_{S_1}, \sigma_{S_1})) \\
\equiv_H (S_2, m_{S_2}(f_{S_2}, \sigma_{S_2})).
\]

If \( s_1 \) and \( s_2 \) start in the state \( m_1(f_1, \text{loop}_c^1, \sigma_1) \) and \( m_2(f_2, \text{loop}_c^2, \sigma_2) \) respectively 
in which crash flags are not set, \( f_1 = f_2 = 0 \), \( s_1 \) and \( s_2 \) have not already executed, 
\( \text{loop}_c^1(n_1) = \text{loop}_c^2(n_2) = 0 \), value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of variables in 
\( TVar(s) \), \( \forall x \in TVar(s), \sigma_1(x) = \sigma_2(x) \), then \( s_1 \) and \( s_2 \) terminate in the same way:

1. \( s_1 \) and \( s_2 \) both terminate, \( (s_1, m_1) \xrightarrow{k} (\text{skip}, m'_1), (s_2, m_2) \xrightarrow{k} (\text{skip}, m'_2) \).
2. \( s_1 \) and \( s_2 \) both do not terminate, \( \forall k > 0, (s_1, m_1) \xrightarrow{k} (S_{1k}, m_{1k}), (s_2, m_2) \xrightarrow{k} (S_{2k}, m_{2k}) \) where \( S_{1k} \neq \text{skip}, S_{2k} \neq \text{skip} \).

This is from Lemma 4.4.12 immediately.

Lemma 4.4.13 is necessary only for showing the same I/O sequence in the next 
section.

Lemma 4.4.13. Let \( s_1 = \"while_{(n_1)}(e)\{S_1\}\) and \\
\( s_2 = \"while_{(n_2)}(e)\{S_2\}\) be two while statements in program \( P_1 \) and \( P_2 \) respectively 
with the same set of termination deciding variables, \( TVar(s_1) = TVar(s_2) = TVar(s) \), 
whose bodies \( S_1 \) and \( S_2 \) satisfy the proof rule of equivalently computation of variables 
in \( TVar(s) \), \( \forall x \in TVar(s) : S_1 \equiv^S_x S_2 \) and whose bodies \( S_1 \) and \( S_2 \) terminate in the
same way in executions when started in states with crash flags not set and agreeing on values of variables in $\text{TVar}(S_1) \cup \text{TVar}(S_2)$:

$$\forall m_{S_1}(f_{S_1}, \sigma_{S_1}) \; m_{S_2}(f_{S_2}, \sigma_{S_2}) :$$

$$((\forall z \in \text{TVar}(S_1) \cup \text{TVar}(S_2)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \land (f_{S_1} = f_{S_2} = 0) \Rightarrow (S_1, m_{S_1}(f_{S_1}, \sigma_{S_1})) \equiv_H (S_2, m_{S_2}(f_{S_2}, \sigma_{S_2})).$$

If $s_1$ and $s_2$ start in the state $m_1(f_1, \text{loop}^1_c, \sigma_1)$ and $m_2(f_2, \text{loop}^2_c, \sigma_2)$ respectively in which crash flags are not set, $f_1 = f_2 = 0$, $s_1$ and $s_2$ have not already executed, $\text{loop}^1_c(n_1) = \text{loop}^2_c(n_2) = 0$, value stores $\sigma_1$ and $\sigma_2$ agree on values of variables in $\text{TVar}(s)$, $\forall x \in \text{TVar}(s), \sigma_1(x) = \sigma_2(x)$, one of the following holds:

1. $s_1$ and $s_2$ both terminate and the loop counters of $s_1$ and $s_2$ are less than a positive integer $i$ and less than or equal to $i - 1$: $(s_1, m_1) \xrightarrow{*} (\text{skip}, m'_1)$, $(s_2, m_2) \xrightarrow{*} (\text{skip}, m'_2)$ where both of the following hold:
   
   - The loop counters of $s_1$ and $s_2$ are less than a positive integer $i$:
     $\exists i > 0 \forall m'_1 \; m'_2$:
     $$(s_1, m_1) \xrightarrow{*} (S'_1, m'_1(\text{loop}^{\prime}_c)), \; \text{loop}^{\prime}_c(n_1) < i$$
     $$(s_2, m_2) \xrightarrow{*} (S'_2, m'_2(\text{loop}^{\prime}_c)), \; \text{loop}^{\prime}_c(n_2) < i.$$ 
   
   - $\forall 0 < j < i$, there are two configurations $(s_1, m_{1_j})$ and $(s_2, m_{2_j})$ reachable from $(s_1, m_1)$ and $(s_2, m_2)$, respectively, in which both crash flags are not set, the loop counters of $s_1$ and $s_2$ are equal to $j$ and value stores agree on the values of variables in $\text{TVar}(s)$, and for every state in execution $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_j})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_j})$, the loop counters for $s_1$ and $s_2$ are less than or equal to $j$ respectively:
     $\exists (s_1, m_{1_j})(s_2, m_{2_j})$:
     $$(s_1, m_1) \xrightarrow{*} (s_1, m_{1_j}(f_1, \text{loop}^{1_j}_c, \sigma_{1_j})) \land$$
     $$(s_2, m_2) \xrightarrow{*} (s_2, m_{2_j}(f_2, \text{loop}^{2_j}_c, \sigma_{2_j}))$$ where
\[ f_1 = f_2 = 0; \text{ and} \]
\[ \text{loop}^{1i}_c(n_1) = \text{loop}^{2i}_c(n_2) = j; \text{ and} \]
\[ \forall x \in \text{TVar}(s) : \sigma_{1i}(x) = \sigma_{2i}(x); \text{ and} \]
\[ \forall m'_1 : (s_1, m_1) \xrightarrow{*} (S'_1, m'_1(\text{loop}^{1'}_c)) \xrightarrow{*} (s_1, m_1), \]
\[ \text{loop}^{1'}_c(n_1) \leq i; \text{ and} \]
\[ \forall m'_2 : (s_2, m_2) \xrightarrow{*} (S'_2, m'_2(\text{loop}^{2'}_c)) \xrightarrow{*} (s_2, m_2), \]
\[ \text{loop}^{2'}_c(n_2) \leq j. \]

2. \( s_1 \) and \( s_2 \) both do not terminate, \( \forall k > 0, (s_1, m_1) \xrightarrow{k} (S_{1k}, m_{1k}), (s_2, m_2) \xrightarrow{k} (S_{2k}, m_{2k}) \) where \( S_{1k} \neq \text{skip}, S_{2k} \neq \text{skip} \) such that one of the following holds:

(a) For any positive integer \( i \), there are two configurations \((s_1, m_{1i})\) and \((s_2, m_{2i})\) reachable from \((s_1, m_1)\) and \((s_2, m_2)\), respectively, in which both crash flags are not set, the loop counters of \( s_1 \) and \( s_2 \) are equal to \( i \) and value stores agree on the values of variables in \( \text{TVar}(s) \), and for every state in execution \((s_1, m_1)i \xrightarrow{*} (s_1, m_{1i})\) or \((s_2, m_2)i \xrightarrow{*} (s_2, m_{2i})\), the loop counters for \( s_1 \) and \( s_2 \) are less than or equal to \( i \) respectively:

\[ \forall i > 0 \exists (s_1, m_{1i}) (s_2, m_{2i}) : \]
\[ (s_1, m_1) \xrightarrow{*} (s_1, m_{1i}(f_1, \text{loop}^{1i}_c, \sigma_{1i})) \wedge \]
\[ (s_2, m_2) \xrightarrow{*} (s_2, m_{2i}(f_2, \text{loop}^{2i}_c, \sigma_{2i})) \]

where

- \( f_1 = f_2 = 0; \) and
- \( \text{loop}^{1i}_c(n_1) = \text{loop}^{2i}_c(n_2) = i; \) and
- \( \forall x \in \text{TVar}(s) : \sigma_{1i}(x) = \sigma_{2i}(x); \) and
- \( \forall m'_1 : (s_1, m_1) \xrightarrow{*} (S'_1, m'_1(\text{loop}^{1'}_c)) \xrightarrow{*} (s_1, m_{1i}), \)
  \[ \text{loop}^{1'}_c(n_1) \leq i; \) and
- \( \forall m'_2 : (s_2, m_2) \xrightarrow{*} (S'_2, m'_2(\text{loop}^{2'}_c)) \xrightarrow{*} (s_2, m_{2i}), \]

60
\[
\text{loop}_c^2(n_2) \leq i;
\]

(b) The loop counters for \(s_1\) and \(s_2\) are less than a positive integer \(i\) and less than or equal to \(i - 1\) such that all of the following hold:

- \(\exists i > 0, \forall m'_1, m'_2 : (s_1, m_1) \stackrel{*}{\rightarrow} (S_1, m'_1(\text{loop}_c^1)), (s_2, m_2) \stackrel{*}{\rightarrow} (S_2, m'_2(\text{loop}_c^2))\) where \(\text{loop}_c^1(n_1) < i, \text{loop}_c^2(n_2) < i\);

- \(\forall 0 < j < i, \text{there are two configurations (}s_1, m_{1j}\text{) and (}s_2, m_{2j}\text{) reachable from (}s_1, m_1\text{) and (}s_2, m_2\text{), respectively, in which both crash flags are not set, the loop counters of }s_1\text{ and }s_2\text{ are equal to }j\text{ and value stores agree on the values of variables in }TVar(s)\text{, and for every state in execution (}s_1, m_1\text{) }\stackrel{*}{\rightarrow} (s_1, m_{1j})\text{ or (}s_2, m_2\text{) }\stackrel{*}{\rightarrow} (s_2, m_{2j})\text{, the loop counters for }s_1\text{ and }s_2\text{ are less than or equal to }j\text{ respectively:}
  \exists(s_1, m_{1j})(s_2, m_{2j}) : \\
  (s_1, m_1) \stackrel{*}{\rightarrow} (s_1, m_{1j}(f_1, \text{loop}_c^{1j}, \sigma_{1j})) \land \\
  (s_2, m_2) \stackrel{*}{\rightarrow} (s_2, m_{2j}(f_2, \text{loop}_c^{2j}, \sigma_{2j}))\text{ where} \\
  - f_1 = f_2 = 0; \text{ and} \\
  - \text{loop}_c^{1j}(n_1) = \text{loop}_c^{2j}(n_2) = j; \text{ and} \\
  - \forall x \in TVar(s) : \sigma_{1j}(x) = \sigma_{2j}(x); \text{ and} \\
  - \forall m'_1 : (s_1, m_1) \stackrel{*}{\rightarrow} (S'_1, m'_1(\text{loop}_c^1)) \stackrel{*}{\rightarrow} (s_1, m_{1j}), \text{ loop}_c^1(n_1) \leq j; \text{ and} \\
  - \forall m'_2 : (s_2, m_2) \stackrel{*}{\rightarrow} (S'_2, m'_2(\text{loop}_c^2)) \stackrel{*}{\rightarrow} (s_2, m_{2j}), \text{ loop}_c^2(n_2) \leq j; \text{ and}

(c) The loop counters for \(s_1\) and \(s_2\) are less than or equal to some positive integer \(i\) such that all of the following hold:

- \(\exists i > 0 \forall m'_1, m'_2 : (s_1, m_1) \stackrel{*}{\rightarrow} (S_1, m'_1(\text{loop}_c^1)), (s_2, m_2) \stackrel{*}{\rightarrow} (S_2, m'_2(\text{loop}_c^2))\) where \(\text{loop}_c^1(n_1) \leq i, \text{loop}_c^2(n_2) \leq i\);
• \(0 < j < i\), there are two configurations \((s_1, m_1)\) and \((s_2, m_2)\) reachable from \((s_1, m_1)\) and \((s_2, m_2)\), respectively, in which both crash flags are not set, the loop counters of \(s_1\) and \(s_2\) are equal to \(j\) and value stores agree on the values of variables in \(TVar(s)\), and for every state in execution \((s_1, m_1) \overset{*}{\rightarrow} (s_1, m_1_j)\) or \((s_2, m_2) \overset{*}{\rightarrow} (s_2, m_2_j)\), the loop counters for \(s_1\) and \(s_2\) are less than or equal to \(j\) respectively:

\[
\exists(s_1, m_1_j)(s_2, m_2) :
(s_1, m_1) \overset{*}{\rightarrow} (s_1, m_1_j(f_1, loop^{1_i}_c, \sigma_1)) \land
(s_2, m_2) \overset{*}{\rightarrow} (s_2, m_2_j(f_2, loop^{2_i}_c, \sigma_2)) \text{ where}
- f_1 = f_2 = 0; and
- loop^{1_i}_c(n_1) = loop^{2_i}_c(n_2) = j; and
- \(\forall x \in TVar(s) : \sigma_{1_j}(x) = \sigma_{2_j}(x); \text{ and}\)
- \(\forall m'_1 : (s_1, m_1) \overset{*}{\rightarrow} (S'_1, m'_1(loop'^{1'}_c)) \overset{*}{\rightarrow} (s_1, m_1_j),\)
  \(loop'^{1'}_c(n_1) \leq j; \text{ and}\)
- \(\forall m'_2 : (s_2, m_2) \overset{*}{\rightarrow} (S'_2, m'_2(loop'^{2'}_c)) \overset{*}{\rightarrow} (s_2, m_2_j),\)
  \(loop'^{2'}_c(n_2) \leq j;\)

• There are no configurations \((s_1, m_1_i)\) and \((s_2, m_2_i)\) reachable from \((s_1, m_1)\) and \((s_2, m_2)\), respectively, in which crash flags are not set, the loop counters of \(s_1\) and \(s_2\) are equal to \(i\), and value stores agree on the values of variables in \(TVar(s)\):

\[
\not\exists(s_1, m_1_i)(s_2, m_2_i) :
(s_1, m_1) \overset{*}{\rightarrow} (s_1, m_1_i(f_1, loop^{1_i}_c, \sigma_1)) \land
(s_2, m_2) \overset{*}{\rightarrow} (s_2, m_2_i(f_2, loop^{2_i}_c, \sigma_2)) \text{ where}
- f_1 = f_2 = 0; and
- loop^{1_i}_c(n_1) = loop^{2_i}_c(n_2) = i; and
- $\forall x \in TVar(s) : \sigma_1(x) = \sigma_2(x)$.

The proof is a case analysis of whether $s_1$ and $s_2$ terminate or not. Refer to our technical report for details [88].

4.5 Behavioral Equivalence

We now propose a proof rule under which two programs produce the same output sequence, namely the same I/O sequence till any $i$th output value. We care about the I/O sequence due to the possible crash from the lack of input. We start by giving the definition of the same output sequence, then we describe the proof rule under which two programs produce the same output sequence, finally we show that our proof rule ensures same output together with the necessary auxiliary lemmas. We use the notation “Out($\sigma$)” to represent the output sequence in value store $\sigma$, the I/O sequence $\sigma(id_{IO})$ till the rightmost output value. Particularly, when there is no output value in the I/O sequence $\sigma(id_{IO})$, Out($\sigma$) = $\emptyset$.

Definition 20. (Same Output Sequence) Two statement sequences $S_1$ and $S_2$ produce the same output sequence when started in states $m_1$ and $m_2$ respectively, written $(S_1, m_1) \equiv_O (S_2, m_2)$, iff $\forall m'_1, m'_2$ such that $(S_1, m_1) \xrightarrow{*} (S'_1, m'_1(\sigma'_1))$ and $(S_2, m_2) \xrightarrow{*} (S'_2, m'_2(\sigma'_2))$, there are states $m''_1, m''_2$ reachable from initial states $m_1$ and $m_2$, $(S_1, m_1) \xrightarrow{*} (S''_1, m''_1(\sigma''_1))$ and $(S_2, m_2) \xrightarrow{*} (S''_2, m''_2(\sigma''_2))$ so that Out($\sigma''_1$) = Out($\sigma'_1$) and Out($\sigma''_2$) = Out($\sigma'_2$).

4.5.1 Proof Rule for Behavioral Equivalence

We show the proof rules of the behavioral equivalence. The output sequence produced in executions of a statement sequence $S$ depends on values of a set of variables in the program, the output deciding variables OVar($S$). The output deciding vari-
ables are of two parts: \( \text{TVar}_o(S) \) are variables affecting the termination of executions of a statement sequence; \( \text{Imp}_o(S) \) are variables affecting values of the I/O sequence produced in executions of a statement sequence. The definitions of \( \text{TVar}_o(S) \) and \( \text{Imp}_o(S) \) are shown in Definition 21 and 22.

**Definition 21. (Imported Variables Relative to Output)** The imported variables in one program \( S \) relative to output, written \( \text{Imp}_o(S) \), are listed as follows:

1. \( \text{Imp}_o(S) = \{id_{IO}\}, \) if \( (\forall e: \text{"output } e\text{" }\notin S) \);

2. \( \text{Imp}_o(\text{"output } e\text{"}) = \{id_{IO}\} \cup \text{Use}(e); \)

3. \( \text{Imp}_o(\text{"If } e\text{ then } \{S_t\} \text{ else } \{S_f\}\text{"}) = \text{Use}(e) \cup \text{Imp}_o(S_t) \cup \text{Imp}_o(S_f) \) if \( (\exists e: \text{"output } e\text{" }\in S) \);

4. \( \text{Imp}_o(\text{"while}_{(n)}(e)\{S''\}) = \text{Imp}(\text{"while}_{(n)}(e)\{S''\}, \{id_{IO}\}) \) if \( (\exists e: \text{"output } e\text{" }\in S'') \);

5. For \( k > 0 \), \( \text{Imp}_o(s_1; \ldots; s_k; s_{k+1}) = \text{Imp}(s_1; \ldots; s_k, \text{Imp}_o(s_{k+1})) \) if \( (\exists e: \text{"output } e\text{" }\in s_{k+1}) \);

6. For \( k > 0 \), \( \text{Imp}_o(s_1; \ldots; s_k; s_{k+1}) = \text{Imp}_o(s_1; \ldots; s_k) \) if \( (\forall e: \text{"output } e\text{" }\notin s_{k+1}) \);

**Definition 22. (Termination Deciding Variables Relative to Output)** The termination deciding variables in a statement sequence \( S \) relative to output, written \( \text{TVar}_o(S) \), are listed as follows:

1. \( \text{TVar}_o(S) = \emptyset \) if \( (\forall e: \text{"output } e\text{" }\notin S) \);

2. \( \text{TVar}_o(\text{"output } e\text{"}) = \text{Err}(e); \)

3. \( \text{TVar}_o(\text{"If } e\text{ then } \{S_t\} \text{ else } \{S_f\}\text{"}) = \text{Use}(e) \cup \text{TVar}_o(S_t) \cup \text{TVar}_o(S_f) \) if \( (\exists e: \text{"output } e\text{" }\in S) \);
4. $\text{TVar}_o(\langle \text{while}_{(n)}(e) \{S''\} \rangle) = \text{TVar}(\langle \text{while}_{(n)}(e) \{S''\} \rangle)$ if ($\exists e : \text{“output } e \text{” } \in S''$);

5. For $k > 0$, $\text{TVar}_o(s_1; \ldots; s_k; s_{k+1}) = \text{TVar}(s_1; \ldots; s_k) \cup \text{Imp}(s_1; \ldots; s_k; \text{TVar}_o(s_{k+1}))$ if ($\exists e : \text{“output } e \text{” } \in s_{k+1}$);

6. For $k > 0$, $\text{TVar}_o(s_1; \ldots; s_k; s_{k+1}) = \text{TVar}_o(s_1; \ldots; s_k)$ if ($\forall e : \text{“output } e \text{” } \notin s_{k+1}$);

**Definition 23. (Output Deciding Variables)** The output deciding variables in a statement sequence $S$ are $\text{Imp}_o(S) \cup \text{TVar}_o(S)$, written $\text{OVar}(S)$.

The condition of the behavioral equivalence is defined recursively. The base case is for two same output statements or two statements where the output sequence variable is not defined. The inductive cases are syntax directed considering the syntax of compound statements and statement sequences.

**Definition 24. (Proof Rule of Behavioral Equivalence)** Two statement sequences $S_1$ and $S_2$ satisfy the proof rule of behavioral equivalence, written $S_1 \equiv^S S_2$, iff one of the following holds:

1. $S_1$ and $S_2$ are one statement and one of the following holds:
   
   (a) $S_1$ and $S_2$ are simple statement and one of the following holds:

   i. $S_1$ and $S_2$ are not output statement, $\forall e_1 e_2$:

   \[ (“\text{output } e_1” \neq S_1) \land (“\text{output } e_2” \neq S_2); \]

   ii. $S_1 = S_2 = “\text{output } e”$.

   (b) $S_1 = \langle \text{If } (e) \text{ then } \{S_1^f \} \text{ else } \{S_1^f \} \rangle$, $S_2 = \langle \text{If } (e) \text{ then } \{S_2^f \} \text{ else } \{S_2^f \} \rangle$ and all of the following hold:
There is an output statement in $S_1$ and $S_2$,

$\exists e_1 e_2 : ("output e_1" \in S_1) \land ("output e_2" \in S_2)$;

$(S_1^f \equiv^S S_2^f) \land (S_1^t \equiv^S S_2^t)$;

(c) $S_1 = \text{"while}_{(n_1)}(e) \{S_1''\}$ and $S_2 = \text{"while}_{(n_2)}(e) \{S_2''\}$ and all of the following hold:

There is an output statement in $S_1$ and $S_2$,

$\exists e_1 e_2 : ("output e_1" \in S_1) \land ("output e_2" \in S_2)$;

$S_1'' \equiv^S S_2''$;

$S_1''$ and $S_2''$ have equivalent computation of $OVar(S_1) \cup OVar(S_2)$;

$S_1''$ and $S_2''$ satisfy the proof rule of termination in the same way, $S_1'' \equiv^S_H S_2''$;

(d) Output statements are not in both $S_1$ and $S_2$,

$\forall e_1 e_2 : ("output e_1" \notin S_1) \land ("output e_2" \notin S_2)$.

2. $S_1$ and $S_2$ are not both one statement and one of the following holds:

(a) $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$, and all of the following hold:

$S_1' \equiv^S S_2'$;

$S_1'$ and $S_2'$ have equivalent computation of $OVar(s_1) \cup OVar(s_2)$;

$S_1'$ and $S_2'$ satisfy the proof rule of termination in the same way: $S_1' \equiv^S_H S_2'$;

There is an output statement in both $s_1$ and $s_2$,

$\exists e_1 e_2 : ("output e_1" \in s_1) \land ("output e_2" \in s_2)$;

$s_1 \equiv^S_O s_2$;

(b) There is no output statement in the last statement in $S_1$ or $S_2$:
\((S_1 = S'_1; s_1) \land (S'_1 \equiv^S_O S_2) \land (\forall e: \text{“output e” } \notin s_1))\)

\(\lor ((S_2 = S'_2; s_2) \land (S_1 \equiv^S_O S'_2) \land (\forall e: \text{“output e” } \notin s_2));\)

4.5.2 Soundness of the Proof Rule for Behavioral Equivalence

We show that two statement sequences satisfy the proof rule of the behavioral equivalence and their initial states agree on values of their output deciding variables, then the two statement sequences produce the same output sequence when started in their initial states.

**Theorem 5.** Two statement sequences \(S_1\) and \(S_2\) satisfy the proof rule of the behavioral equivalence, \(S_1 \equiv^S_O S_2\). If \(S_1\) and \(S_2\) start in states \(m_1(f_1, \sigma_1)\) and \(m_2(f_2, \sigma_2)\) where both of the following hold:

- Crash flags are not set, \(f_1 = f_2 = 0;\)

- Value stores \(\sigma_1\) and \(\sigma_2\) agree on values of the output deciding variables of \(S_1\) and \(S_2\), \(\forall id \in OVar(S_1) \cup OVar(S_2): \sigma_1(id) = \sigma_2(id);\)

then \(S_1\) and \(S_2\) produce the same output sequence, \((S_1, m_1) \equiv^O_O (S_2, m_2).\)

The proof is by induction on the sum of program size of \(S_1\) and \(S_2\), \(\text{size}(S_1) + \text{size}(S_2)\) and is a case analysis based on \(S_1 \equiv^S_O S_2\). Refer to our technical report for details [88].

4.5.3 Supporting Lemmas for the Soundness Proof of Behavioral Equivalence

We listed the lemmas and corollaries used in the proof of Theorem 5 below. The supporting lemmas are of two parts. One part is various properties related to the out-deciding variables. The other part is the proof that two loop statements produce the same output sequence.
Lemma 4.5.1. For any statement sequence $S$, the I/O sequence variable is in imported variable in $S$ relative to output, $id_{IO} \in \text{Imp}_o(S)$.

By structure induction on abstract syntax of $S$. Refer to our technical report for details [88].

Lemma 4.5.2. For any statement sequence $S$, the imported variables in $S$ relative to output are a subset of the imported variables in $S$ relative to the I/O sequence variable, $\text{Imp}_o(S) \subseteq \text{Imp}(S, \{id_{IO}\})$.

By induction on abstract syntax of $S$. Refer to our technical report for details [88].

Lemma 4.5.3. If two statement sequences $S_1$ and $S_2$ satisfy the proof rule of behavioral equivalence, then $S_1$ and $S_2$ have the same set of imported variables relative to output, $(S_1 \equiv^S_{IO} S_2) \Rightarrow (\text{Imp}_o(S_1) = \text{Imp}_o(S_2))$.

By induction on size($S_1$) + size($S_2$). Refer to our technical report for details [88].

Lemma 4.5.4. For any statement sequence $S$ and any variable $x$, the termination deciding variables in $S$ relative to output is a subset of the termination deciding variables in $S$, $T\text{Var}_o(S) \subseteq T\text{Var}(S)$.

By induction on abstract syntax of $S$. Refer to our technical report for details [88].

Lemma 4.5.5. If two statement sequences $S_1$ and $S_2$ satisfy the proof rule of behavioral equivalence, then $S_1$ and $S_2$ have the same set of termination deciding variables relative to output, $(S_1 \equiv^S_{IO} S_2) \Rightarrow (T\text{Var}_o(S_1) = T\text{Var}_o(S_2))$.

Proof. By induction on size($S_1$) + size($S_2$). \hfill \Box

Corollary 4.5.1. If two statement sequences $S_1$ and $S_2$ satisfy the proof rule of behavioral equivalence, then $S_1$ and $S_2$ have the same set of out-deciding variables, $(S_1 \equiv^S_{IO} S_2) \Rightarrow O\text{Var}(S_1) = O\text{Var}(S_2)$. 

68
Proof. By Lemma 4.5.3, $\text{Imp}_o(S_1) = \text{Imp}_o(S_2)$. By Lemma 4.5.5, $\text{TVar}_o(S_1) = \text{TVar}_o(S_2)$.

Lemma 4.5.6. In one step execution $(S, m(\sigma)) \rightarrow (S', m'(\sigma'))$, if there is no output statement in $S$, then the output sequence is same in value store $\sigma$ and $\sigma'$, $\text{Out}(\sigma') = \text{Out}(\sigma)$.

Proof. By induction on abstract syntax of $S$ and crash flag $f$ in state $m$.

Lemma 4.5.7. If there is no output statement in $S$, then, after the execution $(S, m(\sigma)) \xrightarrow{k} (S', m'(\sigma'))$. The proof also relies on the fact that if $s \notin S$, then $s \notin S'$.

Lemma 4.5.8. One while statement $s = \text{"while}_{(n)}(e)\{S\}$ starts in a state $m(f, \text{loop}_c)$ in which the loop counter of $s$ is zero, $\text{loop}_c(n) = 0$ and the crash flag is not set, $f = 0$.

For any positive integer $i$, if there is a state $m'(m'_c)$ reachable from $m$ in which the loop counter is $i$, $\text{loop}'_c(n) = i$, then there is a configuration $(S; s, m''(f'', \text{loop}_c''))$ reachable from the configuration $(s, m)$ in which loop counter of $s$ is $i$, $\text{loop}_c''(n) = i$ and the crash flag is not set, $f'' = 0$.

The proof is by induction on $i$. Refer to our technical report for details [88].

Lemma 4.5.9. Let $s_1 = \text{"while}_{(n_1)}(e)\{S_1\}$ and $s_2 = \text{"while}_{(n_2)}(e)\{S_2\}$ be two while statements and all of the followings hold:

- There are output statements in $s_1$ and $s_2$,
- $\exists e_1 e_2 : (\text{"output } e_1 \text{" } \in s_1) \land (\text{"output } e_2 \text{" } \in s_2)$;
s_1 and s_2 have the same set of termination deciding variables relative to output, and the same set of imported variables relative to output, (TVar_o(s_1) = TVar_o(s_2) = TVar(s)) ∧ (Imp_o(s_1) = Imp_o(s_2) = Imp(io));

Loop bodies S_1 and S_2 satisfy the proof rule of equivalent computation of the output deciding variables of s_1 and s_2, ∀x ∈ OVar(s) = TVar(s) ∪ Imp(io) : S_1 ≡_x^S S_2;

Loop bodies S_1 and S_2 satisfy the proof rule of termination in the same way, S_1 ≡_H^S S_2;

Loop bodies S_1 and S_2 produce the same output sequence when started in states with crash flags not set and whose value stores agree on values of variables in OVar(S_1) ∪ OVar(S_2), ∀m_{s_1}(f_1, σ_{s_1}) m_{s_2}(f_2, σ_{s_2}) :

((f_1 = f_2 = 0) ∧ (∀x ∈ OVar(S_1) ∪ OVar(S_2) : σ_{s_1}(x) = σ_{s_2}(x))) ⇒

((S_1, m_{s_1}(f_1, σ_{s_1})) ≡_O (S_2, m_{s_2}(f_2, σ_{s_2}))).

If s_1 and s_2 start in states m_1(f_1, loop_c^1, σ_1), m_2(f_2, loop_c^2, σ_2) respectively with crash flags not set f_1 = f_2 = 0 and in which s_1 and s_2 have not started execution (loop_c^1(n_1) = loop_c^2(n_2) = 0), value stores σ_1 and σ_2 agree on values of variables in OVar(s), ∀x ∈ OVar(s) : σ_1(x) = σ_2(x), then one of the followings holds:

1. s_1 and s_2 both terminate and produce the same output sequence:

   (s_1, m_1) → (skip, m'_1(σ'_1)), (s_2, m_2) → (skip, m'_2(σ'_2)) where σ'_1(id_{IO}) = σ'_2(id_{IO}).

2. s_1 and s_2 both do not terminate, ∀k > 0, (s_1, m_1) →^k (S_{1k}, m_{1k}), (s_2, m_2) →^k (S_{2k}, m_{2k}) where S_{1k} ≠ skip, S_{2k} ≠ skip and one of the followings holds:

   (a) For any positive integer i, there are two configurations (s_1, m_{1i}) and (s_2, m_{2i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to i and value stores
agree on values of variables in OVar(s), and for every state in execution, 
\((s_1, m_1) \to (s_1, m_1)\) or \((s_2, m_2) \to (s_2, m_2)\), loop counters for \(s_1\) and \(s_2\) are less than or equal to \(i\) respectively:
\[\forall i > 0 \exists (s_1, m_1), (s_2, m_2) : (s_1, m_1) \to (s_1, m_1, (f_1, \text{loop}_{1}^i, \sigma_{1})), (s_2, m_2) \to (s_2, m_2, (f_2, \text{loop}_{2}^i, \sigma_{2})),\]
where
- \(f_1 = f_2 = 0\); and
- \(\text{loop}_{1}^i(n_1) = \text{loop}_{2}^i(n_2) = i\); and
- \(\forall x \in \text{OVar}(s) : \sigma_{1}(x) = \sigma_{2}(x)\).
- \(\forall m_1' : (s_1, m_1) \to (S_1', m_1'(\text{loop}_{1}^i)), (s_1, m_1, (f_1, \text{loop}_{1}^i, \sigma_{1})), \text{loop}_{1}'(n_1) \leq i\); and
- \(\forall m_2' : (s_2, m_2) \to (S_2', m_2'(\text{loop}_{2}^i)), (s_2, m_2, (f_2, \text{loop}_{2}^i, \sigma_{2})), \text{loop}_{2}'(n_2) \leq i\).
(b) The loop counters for \(s_1\) and \(s_2\) are less than a smallest positive integer \(i\) and all of the followings hold:
- \(\exists i > 0 \forall m_1', m_2' : (s_1, m_1) \to (S_1', m_1'(\text{loop}_{1}^i)), (s_2, m_2) \to (S_2', m_2'(\text{loop}_{2}^i))\) where \(\text{loop}_{1}'(n_1) < i, \text{loop}_{2}'(n_2) < i\);
- \(\forall 0 < j < i\), there are two configurations \((s_1, m_1)\) and \((s_2, m_2)\) reachable from \((s_1, m_1)\) and \((s_2, m_2)\), respectively, in which both crash flags are not set, loop counters of \(s_1\) and \(s_2\) are equal to \(j\) and value stores agree on values of variables in \(OVar(s)\):
\[\exists (s_1, m_{1j}), (s_2, m_{2j}) : (s_1, m_1) \to (s_1, m_{1j}, (f_1, \text{loop}_{1}^j, \sigma_{1})), (s_2, m_2) \to (s_2, m_{2j}, (f_2, \text{loop}_{2}^j, \sigma_{2})),\]
where
- \(f_1 = f_2 = 0\); and
- \(\text{loop}_{1}^j(n_1) = \text{loop}_{2}^j(n_2) = j\); and
- \(\forall x \in \text{OVar}(s) : \sigma_{1j}(x) = \sigma_{2j}(x)\).
• If \( i = 1 \), then the I/O sequence is not redefined in any states reachable from \((s_1, m_1)\) and \((s_2, m_2)\).
  
  - \( \forall m''_1 : (s_1, m_1(\text{loop}^1_c, \sigma_1)) \xrightarrow{*} (S''_1, m''_1(\sigma''_1)) \) where \( \sigma''_1(id_{IO}) = \sigma_1(id_{IO}) \).
  
  - \( \forall m''_2 : (s_2, m_2(\text{loop}^2_c, \sigma_2)) \xrightarrow{*} (S''_2, m''_2(\sigma''_2)) \) where \( \sigma''_2(id_{IO}) = \sigma_2(id_{IO}) \).

• If \( i > 1 \), then the I/O sequence is not redefined in any states reachable from \((s_1, m_{1_{i-1}})\) and \((s_2, m_{2_{i-1}})\).
  
  - \( \forall m''_1 : (s_1, m_{1_{i-1}}(\text{loop}^{1_{i-1}}_c, \sigma_{1_{i-1}})) \xrightarrow{*} (S''_1, m''_1(\sigma''_1)) \) where \( \sigma''_1(id_{IO}) = \sigma_{1_{i-1}}(id_{IO}) \).
  
  - \( \forall m''_2 : (s_2, m_{2_{i-1}}(\text{loop}^{2_{i-1}}_c, \sigma_{2_{i-1}})) \xrightarrow{*} (S''_2, m''_2(\sigma''_2)) \) where \( \sigma''_2(id_{IO}) = \sigma_{2_{i-1}}(id_{IO}) \).

(c) The loop counters for \( s_1 \) and \( s_2 \) are less than or equal to a smallest positive integer \( i \) and all of the followings hold:

• \( \exists i > 0 \forall m'_1, m'_2 : (s_1, m_1) \xrightarrow{*} (S'_1, m'_1(\text{loop}^{1'}_c)), (s_2, m_2) \xrightarrow{*} (S'_2, m'_2(\text{loop}^{2'}_c)) \) where \( \text{loop}^{1'}_c(n_1) \leq i, \text{loop}^{2'}_c(n_2) \leq i \);

• There are no configurations \((s_1, m_{1_{i}})\) and \((s_2, m_{2_{i}})\) reachable from \((s_1, m_1)\) and \((s_2, m_2)\), respectively, in which crash flags are not set, the loop counters of \( s_1 \) and \( s_2 \) are equal to \( i \), and value stores agree on values of variables in \( \text{OVar}(s) \):
  
  - \( \#(s_1, m_{1_{i}}), (s_2, m_{2_{i}}) : (s_1, m_1) \xrightarrow{*} (s_1, m_{1_{i}}(f_1, \text{loop}^{1_{i}}_c, \sigma_{1_{i}})) \) \& \( (s_2, m_{2_{i}}(f_2, \text{loop}^{2_{i}}_c, \sigma_{2_{i}})) \) where
    
    - \( f_1 = f_2 = 0 \); and
    
    - \( \text{loop}^{1_{i}}_c(n_1) = \text{loop}^{2_{i}}_c(n_2) = i \); and
    
    - \( \forall x \in \text{OVar}(s) : \sigma_{1_{i}}(x) = \sigma_{2_{i}}(x) \).

• \( \forall 0 < j < i \), there are two configurations \((s_1, m_{1_{j}})\) and \((s_2, m_{2_{j}})\) reachable from \((s_1, m_{1_{i}})\) and \((s_2, m_{2_{i}})\), respectively, in which crash flags are
not set, the loop counters of \( s_1 \) and \( s_2 \) are equal to \( j \) and value stores agree on values of variables in \( \text{OVar}(s) \):

\[
\exists (s_1, m_{1_j}), (s_2, m_{2_j}) : (s_1, m_1) \xrightarrow{*} (s_1, m_{1_j}(f_1, \text{loop}^1_c, \sigma_{1_j})) \land (s_2, m_2) \xrightarrow{*} (s_2, m_{2_j}(f_2, \text{loop}^2_c, \sigma_{2_j}))
\]

where

- \( f_1 = f_2 = 0 \); and
- \( \text{loop}^1_c(n_1) = \text{loop}^2_c(n_2) = j \); and
- \( \forall x \in \text{OVar}(s) : \sigma_{1_j}(x) = \sigma_{2_j}(x) \).

- **If** \( i = 1 \), then executions from \( (s_1, m_1) \) and \( (s_2, m_2) \) produce the same output sequence:

\[
(s_1, m_1(\text{loop}^1_c, \sigma_1)) \equiv_O (s_2, m_2(\text{loop}^2_c, \sigma_2)).
\]

- **If** \( i > 1 \), then executions from \( (s_1, m_{1_{i-1}}) \) and \( (s_2, m_{2_{i-1}}) \) produce the same output sequence:

\[
(s_1, m_{1_{i-1}}(\text{loop}^1_{c_{i-1}}, \sigma_{1_{i-1}})) \equiv_O (s_2, m_{2_{i-1}}(\text{loop}^2_{c_{i-1}}, \sigma_{2_{i-1}})).
\]

We show that \( s_1 \) and \( s_2 \) terminate in the same way when started in states \( m_1(f_1, \text{loop}^1_c, \sigma_1) \) and \( m_2(f_2, \text{loop}^2_c, \sigma_2) \) respectively, \( (s_1, m_1) \equiv_H (s_2, m_2) \). In addition, we show that \( s_1 \) and \( s_2 \) produce the same output sequence in every possibilities of termination in the same way, \( (s_1, m_1) \equiv_O (s_2, m_2) \). Refer to our technical report for details [88].

**Corollary 4.5.2.** Let \( s_1 = \text{“while}_{(n_1)}(e) \{S_1\}” \) and \( s_2 = \text{“while}_{(n_2)}(e) \{S_2\}” \) be two while statements such that all of the followings hold

- There are output statements in \( s_1 \) and \( s_2 \), \( \exists e_1, e_2 : (\text{“output } e_1 \text{” } \in s_1) \land (\text{“output } e_2 \text{” } \in s_2) \);

- \( s_1 \) and \( s_2 \) have same set of termination deciding variables and same set of imported variables relative to the I/O sequence variable, \( (\text{TVar}(s_1) = \text{TVar}(s_2) = \text{TVar}(s)) \land (\text{Imp}(s_1, \{id_{IO}\}) = \text{Imp}(s_2, \{id_{IO}\}) = \text{Imp}(io)) \);
• Loop bodies $S_1$ and $S_2$ satisfy the proof rule of equivalent computation of those in out-deciding variables of $s_1$ and $s_2$, $\forall x \in OVar(s) = TVar(s) \cup Imp(io)$ : $S_1 \equiv_x^S S_2$;

• Loop bodies $S_1$ and $S_2$ satisfy the proof rule of termination in the same way, $S_1 \equiv^S H S_2$;

• Loop bodies $S_1$ and $S_2$ produce the same output sequence when started in states with crash flags not set and agreeing on values of variables in $OVar(S_1) \cup OVar(S_2)$, $\forall m_{S_1}(f_1, \sigma_{S_1}) m_{S_2}(f_2, \sigma_{S_2}) :$

$((f_1 = f_2 = 0) \land (\forall x \in OVar(S_1) \cup OVar(S_2) : \sigma_{S_1}(x) = \sigma_{S_2}(x))) \Rightarrow ((S_1, m_{S_1}(f_1, \sigma_{S_1})) \equiv_O S_2, m_{S_2}(f_2, \sigma_{S_2})))$.

If $s_1$ and $s_2$ start in states $m_1(f_1, \text{loop}_1^c, \sigma_1), m_2(f_2, \text{loop}_2^c, \sigma_2)$ respectively with crash flags not set $f_1 = f_2 = 0$ and in which $s_1$ and $s_2$ have not started execution ($\text{loop}_1^c(n_1) = \text{loop}_2^c(n_2) = 0$), value stores $\sigma_1$ and $\sigma_2$ agree on values of variables in $OVar(s)$, $\forall x \in OVar(s) : \sigma_1(x) = \sigma_2(x)$, then $s_1$ and $s_2$ produce the same output sequence: $(s_1, m_1) \equiv_O (s_2, m_2)$.

This is from lemma 4.5.9.

4.6 Backward Compatibility of Equivalent Programs

Based on the equivalence result above, we show that there exists backward compatible DSU. We need to show there exists a mapping of old program configurations and new program configurations and the hybrid execution obtained from the configuration mapping is backward compatible. We do not provide a practical algorithm to calculate the state mapping. Instead we only show that there exist new program
configurations corresponding to some old program configurations via a simulation. The treatment in this section is informal.

The idea is to map a configuration just before an output is produced to a corresponding configuration. Based on the proof rule of same output sequences, not every statement of the old program can correspond to a statement of the new program, but every output statement of the old program should correspond to an output statement of the new program. Consider configuration $C_1$ of the old program where the leftmost statement (next statement to execute) is an output statement. We can define a corresponding statement of the new program by simulating the execution of the new program on the input consumed so far in $C_1$. There are two cases. When the leftmost statement in $C_1$ is not included in a loop statement, then it is easy to know when to stop simulation. Otherwise, we have the bijection of loop statements including output statements based on the condition of same output sequences. Therefore, it is easy to know how many iterations of the loop statements including the output statement shall be carried out based on the loop counters in the old program configuration $C_1$. Based on Theorem 5, there must be a configuration $C_2$ corresponding to $C_1$. Moreover, the executions starting from configurations $C_1$ and $C_2$ produce the same output sequence based on Theorem 5. In conclusion, we obtain a backward compatible hybrid execution where the state mapping is from $C_1$ to $C_2$. 
Chapter 5

REAL WORLD BACKWARD COMPATIBLE PROGRAM UPDATES

We present our study of the real world program evolution. From the studied program evolution, we summarize classifications of backward compatible updates. Then we propose our formal treatment for real world update classes. For each update class, we show how an old program and a new program produce the same I/O sequence which guarantees backward compatible DSU.

5.1 Summary of a Study of Real World Program Evolution

We introduce classes of backward compatible program updates that are summarized from our study of real world program updates. The classes of backward compatible updates make it possible to automate state mapping in presence of update classes.

We have studied evolution of three real world programs (i.e., vsftpd, sshd and icecast) to identify real world changes that are backward compatible. We chose these three programs because the programs are widely used in practice [12, 11] and are widely studied in the DSU community [67, 74]. We have studied several years of releases of vsftpd and consecutive updates of sshd and icecast. This is because vsftpd is more widely studied by the DSU community [67, 74, 66].

Our study of real world program evolution is carried out as follows. We examined every changed function manually to classify updates. For every individual change, we first identified the motivation of the change, then the assumptions under which the change could be considered backward compatible. If the assumption under which the change is considered backward compatible is reasonable, we recorded the change into
one particular update class. Finally we summarized common update classes observed in the evolution of studied programs.

Fig. 5.1 and 5.2 shows the statistics from our study of real world program evolution where “total” refers to the number of all updated functions, “class” refers to the number of updated functions with at least one classified update pattern. In summary, 32% of all updated functions includes at least one classified program update; the unclassified updates are mostly bug fixes that are related to specific program logic. We summarized seven most common real world update classes from all the studied updates in Fig. 5.3 and we believe that these update classes are also widespread in other program evolution. Each of the six real world update classes falls in one of the five cases of backward compatibility in Fig. 3.1. We present informal descriptions of all update classes including required assumptions for the two programs to produce same or equivalent output sequence which guarantees backward compatible DSU.

5.1.1 Observational Equivalence: the Old Behavior

In case 1 in Fig. 3.1, two programs are backward compatible because the new program keeps all old behaviors (“observational equivalence”). In our study, we differentiate two types of “observational equivalence” based on if assumptions are required.

Program Equivalence We consider several types of program changes that are allowed by “observational equivalence” without user assumptions. These changes include: loop fission or fusion, statement reordering or duplication, and extra statements unrelated to output (e.g., logging related changes). We incorporate these changes in our framework of program equivalence which ensures two programs produce the same output regardless of whether the programs terminate or not. The details of the formal treatment is in Chapter 4.
<table>
<thead>
<tr>
<th>Software Version</th>
<th>Update Date</th>
<th>Total</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsftpd 1.1.0 – 1.1.1</td>
<td>2002-10-07</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>vsftpd 1.1.1 – 1.1.2</td>
<td>2002-10-16</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>vsftpd 1.1.2 – 1.1.3</td>
<td>2002-11-09</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>vsftpd 1.1.3 – 1.2.0</td>
<td>2003-05-29</td>
<td>61</td>
<td>9</td>
</tr>
<tr>
<td>vsftpd 1.2.0 – 1.2.1</td>
<td>2003-11-13</td>
<td>33</td>
<td>11</td>
</tr>
<tr>
<td>vsftpd 1.2.1 – 1.2.2</td>
<td>2004-04-26</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>vsftpd 1.2.2 – 2.0.0</td>
<td>2004-07-01</td>
<td>52</td>
<td>13</td>
</tr>
<tr>
<td>vsftpd 2.0.0 – 2.0.1</td>
<td>2004-07-02</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>vsftpd 2.0.1 – 2.0.2</td>
<td>2005-03-03</td>
<td>23</td>
<td>4</td>
</tr>
<tr>
<td>vsftpd 2.0.2 – 2.0.3</td>
<td>2005-03-19</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>vsftpd 2.0.3 – 2.0.4</td>
<td>2006-01-09</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>vsftpd 2.0.4 – 2.0.5</td>
<td>2006-07-03</td>
<td>21</td>
<td>15</td>
</tr>
<tr>
<td>vsftpd 2.0.5 – 2.0.6</td>
<td>2008-02-13</td>
<td>20</td>
<td>9</td>
</tr>
<tr>
<td>vsftpd 2.0.6 – 2.0.7</td>
<td>2008-07-30</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>vsftpd 2.0.7 – 2.1.0</td>
<td>2009-02-19</td>
<td>53</td>
<td>11</td>
</tr>
<tr>
<td>vsftpd 2.1.0 – 2.1.2</td>
<td>2009-05-29</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td>vsftpd 2.1.2 – 2.2.0</td>
<td>2009-08-13</td>
<td>34</td>
<td>14</td>
</tr>
</tbody>
</table>

Figure 5.1: Statistics of Classified Real World Software Update - Part 1
<table>
<thead>
<tr>
<th>Software Version</th>
<th>Update Date</th>
<th>Total</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsftpd 2.2.0 – 2.2.2</td>
<td>2009-10-19</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>vsftpd 2.2.2 – 2.3.0</td>
<td>2010-08-06</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>vsftpd 2.3.0 – 2.3.2</td>
<td>2010-08-19</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>vsftpd 2.3.2 – 2.3.4</td>
<td>2011-03-12</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>vsftpd 2.3.4 – 2.3.5</td>
<td>2011-12-19</td>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>vsftpd 2.3.5 – 3.0.0</td>
<td>2012-04-10</td>
<td>23</td>
<td>4</td>
</tr>
<tr>
<td>vsftpd 3.0.0 – 3.0.2</td>
<td>2012-09-19</td>
<td>40</td>
<td>2</td>
</tr>
<tr>
<td>sshd 3.5p1 – 3.6p1</td>
<td>2003-03-31</td>
<td>95</td>
<td>34</td>
</tr>
<tr>
<td>sshd 3.6p1 – 3.6.1p1</td>
<td>2003-04-01</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>sshd 3.6.1p1 – 3.6.1p2</td>
<td>2003-04-29</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>sshd 4.5p1 – 4.6p1</td>
<td>2007-03-07</td>
<td>48</td>
<td>13</td>
</tr>
<tr>
<td>sshd 6.6p1 – 6.7p1</td>
<td>2014-10-06</td>
<td>283</td>
<td>51</td>
</tr>
<tr>
<td>icecast 0.8.0 – 0.8.1</td>
<td>2004-08-04</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>icecast 0.8.1 – 0.8.2</td>
<td>2004-08-04</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>icecast 2.3.0 – 2.3.1</td>
<td>2005-11-30</td>
<td>47</td>
<td>10</td>
</tr>
<tr>
<td>icecast 2.3.1 – 2.3.2</td>
<td>2008-06-02</td>
<td>250</td>
<td>28</td>
</tr>
<tr>
<td>icecast 2.4.0 – 2.4.1</td>
<td>2014-11-19</td>
<td>178</td>
<td>154</td>
</tr>
</tbody>
</table>

Figure 5.2: Statistics of Classified Real World Software Update - Part2
<table>
<thead>
<tr>
<th>Update Class (Case)</th>
<th>Required Assumptions for Backward Compatible Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>program equivalence (1)</td>
<td>none</td>
</tr>
<tr>
<td>new config. variables (1)</td>
<td>no redefinitions of new config variables after initialization</td>
</tr>
<tr>
<td>enum type extension (2)</td>
<td>no inputs from old clients match the extended enum labels</td>
</tr>
<tr>
<td>var. type weakening (3)</td>
<td>no intentional use of value type mismatch and array out of bound</td>
</tr>
<tr>
<td>exit on error (4)</td>
<td>correct error check before exit</td>
</tr>
<tr>
<td>improved prompt msgs (5)</td>
<td>changing prompt messages for more effective communication</td>
</tr>
<tr>
<td>missing var. init. (6)</td>
<td>no intentional use of undefined variables</td>
</tr>
</tbody>
</table>

Figure 5.3: Required Assumptions for Real World Backward Compatible Update Classes

**Specializing New Configuration Variables** Another update class of “observational equivalence” is “specializing new configuration variables”, which is backward compatible under user assumptions. In this update class, new configuration variables are introduced to generalize functionality. For example, in Fig. 5.4, a new configuration variable \( b \) is used to introduce new code. The two statement sequences in Fig. 5.4 are equivalent when the new variable \( b \) is specialized to 0. In general, if all new code is introduced in a way that is similar to that in Fig. 5.4 where there is a valuation of new configuration variables under which new code is not executed, and new configu-
ration variables are not redefined after initialization, then the new program and the old program produce the same output sequence. The point is that new functionality is not introduced abruptly in interaction with an old client. Instead new functionality could be enabled for a new client when old clients are not a concern.

5.1.2 Enumeration Type Extension: Old Behavior for Old Input and Allowing New Input

Enumeration types allow developers to list similar items. New code is usually accompanied with the introduction of new enumeration labels. Fig. 5.5 shows an example of the update. The new enum label \( o_2 \) gives a new option for matching the value of the variable \( a \), which introduces the new code “\( \text{output} \ 3 + c \)”. To show enumeration type extensions to be backward compatible, we assume that values of enum variables, used in the If-predicate introducing the new code, are only from inputs that cannot be translated to new enum labels. This is case 2 of the backward compatibility.

5.1.3 Variable Type Weakening: More Output When the Old Program Terminates

In program updates, variable types are changed either to allow for larger ranges (weakening) or smaller ranges to save space (strengthening). For example, an integer
1: `enum id {o_1}`
1’: `enum id {o_1, o_2}`
2: `a : enum id`
2’: `a : enum id`
3: `If (a == o_1) then`
3’: `If (a == o_1) then`
4: `output 2 + c`
4’: `output 2 + c`
5: 
5’: `If (a == o_2) then`
6: 
6’: `output 3 + c`

Old
New

Figure 5.5: Enumeration Type Extension

variable might be changed to become a long variable to avoid integer overflow or a long variable might be changed to an integer variable because the larger range of long is not needed. Type weakening also includes adding a new enumeration value and increasing array size. The kinds of strengthening or weakening that should be allowed are application dependent and would need to be defined by the user in general. The type weakening considered is either changes from type int to long or increase of array size. These updates fix integer overflow or array index out of bound respectively, the case 3 of backward compatibility. Implicitly, we assume that there is no intentional use of integer overflow and array out of bound as program semantics.

5.1.4 Exit on Errors: Stopping Execution While the Old Program Produces More Output

One kind of bug fix, which we call exit on error, causes a program to exit in observation of errors that depend on application semantic. Fig. 5.6 shows an example of exit-on-error update. In the example, the fixed bugs refer to the program semantic error that `a = 5`. Instead of using an “exit” statement, we rely on the crash from expression evaluations to model the “exit”. When errors do not occur, the two pro-
grams in Fig. 5.6 produce the same output sequence. This is case 4 of backward compatibility. Naturally, we assume that all error checks are correct.

### 5.1.5 Improved Prompt Messages: Functionally Equivalent Outputs

In practice, outputs could be classified into prompt outputs and actual outputs. Prompt outputs are those asking clients for inputs, which are constants hardcoded in output statements. Actual outputs are dynamic messages produced by evaluation of non-constant expressions in execution. If the differences between two programs are only the prompt messages that a client receives, we consider that the two programs are equivalent. The prompt messages are the replaceable part of program semantics. We observe cases of improving prompt messages in program evolution for effective communication. The changes of prompt outputs do not matter only for human clients. This is case 5 of backward compatibility.

### 5.1.6 Missing Variable Initialization: Enforcing Restrictions on Program States

Another kind of bug fix, which we call *missing variable initialization*, includes initializations for variables whose arbitrary initial values can affect the output sequence in the old program. Fig. 5.7 shows an example of missing variable initialization. The initialization $b := 2$ ensures the value used in “output $b + c$” not to be unde-
fined. Despite of initialization statements, the two programs are same. In general, initializations of variables only affect rare buggy executions of the old program, where undefined variables affect the output sequence. This update class is case 6 of backward compatibility and we assume that there is no intentional use of undefined variable in the program. When there are no uses of variables with undefined variables in executions of the old program, the two programs produce the same output sequence.

5.2 Proof Rule for Specializing New Configuration Variables

New configuration variables can be introduced to generalize functionality. Figure 5.4 shows an example of how a new configuration variable introduces new code. The two statement sequences in Figure 5.4 are equivalent when the new variable b is specialized to 0.

Our generalized formal definition of “specializing new configuration variables” is defined as follows.

**Definition 25. (Specializing New Configuration Variables)** A statement sequence $S_2$ includes updates of specializing new configuration variables compared with $S_1$ w.r.t a mapping $\rho$ of new configuration variables in $S_2$, $\rho : \{id\} \mapsto \{0, 1\}$, denoted $S_2 \approx_\rho S_1$, iff one of the following holds:
1. $S_2 = \text{"If}(\text{id})\text{ then}\{S'_2\} \text{ else}\{S''_2\}$ where one of the following holds:

   (a) $(\rho(\text{id}) = 0) \land (S'_2 \approx^S_{\rho} S_1)$;

   (b) $(\rho(\text{id}) = 1) \land (S'_2 \approx^S_{\rho} S_1)$;

2. $S_1$ and $S_2$ produce the same output sequence, $S_1 \approx^S_{\rho} S_2$;

3. $S_1 = \text{"If}(e)\text{ then}\{S'_1\} \text{ else}\{S''_1\}$, $S_2 = \text{"If}(e)\text{ then}\{S'_2\} \text{ else}\{S''_2\}$ where

   $(S'_1 \approx^S_{\rho} S'_1) \land (S'_2 \approx^S_{\rho} S'_2)$;

4. $S_1 = \text{"while}_{(n_1)}(e)\{S'_1\}$, $S_2 = \text{"while}_{(n_2)}(e)\{S'_2\}$ where

   $S'_2 \approx^S_{\rho} S'_1$;

5. $S_1 = S'_1; s_1$ and $S_2 = S'_2; s_2$ where $(S'_2 \approx^S_{\rho} S'_1) \land (S'_2 \approx^S_{\rho} S'_1) \land (\forall x \in \text{Imp}(s_1, id_{IO}) \cup \text{Imp}(s_1, id_{IO}) : (S'_2 \approx^x_{s} S'_1)) \land (s_2 \approx^S_{\rho} s_1)$.

Then we show that executions of two statement sequences produce the same I/O sequence if there are updates of specializing new configuration variables between the two.

**Lemma 5.2.1.** Let $S_1$ and $S_2$ be two different statement sequences where there are updates of “specializing new configuration variables” in $S_2$ compared with $S_1$ w.r.t a mapping of new configuration variables $\rho$, $S_2 \approx^S_{\rho} S_1$. If executions of $S_2$ and $S_1$ start in states $m_2(f_2, \sigma_2)$ and $m_1(f_1, \sigma_1)$ respectively where all of the following hold:

- **Crash flags** $f_2, f_1$ are not set, $f_2 = f_1 = 0$;

- **Value stores** $\sigma_1$ and $\sigma_2$ agree on output deciding variables in both $S_1$ and $S_2$ including the input and I/O sequence variable,

  $\forall id \in (OVar(S_1) \cap OVar(S_2)) \cup \{id_I, id_{IO}\} : \sigma_1(id) = \sigma_2(id)$;
• Values of new configuration variables in the value store $\sigma_2$ are matching those in $\rho$, $\forall \text{id} \in \text{Dom}(\rho) : \rho(\text{id}) = \sigma_2(\text{id})$;

• Values of new configuration variables are not defined in the statement sequence $S_2$, $\text{Dom}(\rho) \cap \text{Def}(S_2) = \emptyset$;

then $S_2$ and $S_1$ satisfy all of the following:

• $(S_1, m_1) \equiv_H (S_2, m_2)$;

• $(S_1, m_1) \equiv_O (S_2, m_2)$;

• $\forall x \in \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;

The proof of Lemma 5.2.1 is by induction on the sum of program sizes of $S_1$ and $S_2$ and is a case analysis based on Definition 25. Refer to our technical report for details [88].

We list properties of the update of new configuration variables and the proof of backward compatibility for the case of loop statement as follows. We present one auxiliary lemma used in the proof of Lemma 5.2.1.

**Lemma 5.2.2.** Let $S_2$ be a statement sequence and $S_1$ where there are updates of “specializing new configuration variables” w.r.t a mapping of new configuration variables $\rho$, $S_2 \approx^S S_1$. Then the output deciding variables in $S_1$ are a subset of the union of those in $S_2$, $\text{OVar}(S_1) \subseteq \text{OVar}(S_2)$.

**Proof.** By induction on the sum of the program size of $S_1$ and $S_2$. \qed

**Lemma 5.2.3.** Let $S_1 = \text{while}_{(n_1)}(e)\{S_1'\}$ and $S_2 = \text{while}_{(n_2)}(e)\{S_2'\}$ be two loop statements where all of the following hold:

• $S_2'$ includes updates of “specializing new configuration variables” compared to $S_1'$, $S_2' \approx^S S_1'$ where $\text{Dom}(\rho) \cap \text{Def}(S_2') = \emptyset$.  

86
the output deciding variables in \( S_1 \) are a subset of those in \( S_2 \),
\[
OVar(S_1) \subseteq OVar(S_2);
\]

When started in states agreeing on values of output deciding variables in \( S_1 \) and \( S_2 \) including the input sequence variable and the I/O sequence variable, \( \forall x \in OVar(S_1) \cup OVar(S_2) \cup \{id_I, id_{IO}\} \forall m'_1(\sigma'_1) m'_2(\sigma'_2) : (\sigma'_1(x) = \sigma'_2(x)) \), \( S'_1 \) and \( S'_2 \) terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in \( S'_1 \) and \( S'_2 \) as well as the input sequence variable and the I/O sequence variable \( ((S'_1, m_1) \equiv_H (S'_2, m_2)) \land ((S'_1, m_1) \equiv_O (S'_2, m_2)) \land (\forall x \in OVar(S_1) \cup OVar(S_2) \cup \{id_I, id_{IO}\} : (S'_1, m_1) \equiv_x (S'_2, m_2)) \);

If \( S_1 \) and \( S_2 \) start in states \( m_1(\text{loop}_c^1, \sigma_1), m_2(\text{loop}_c^2, \sigma_2) \) respectively, with loop counters of \( S_1 \) and \( S_2 \) not initialized \( (S_1, S_2 \text{ have not executed yet}), \) value stores agree on values of output deciding variables in \( S_1 \) and \( S_2 \), then, for any positive integer \( i \), one of the following holds:

1. Loop counters for \( S_1 \) and \( S_2 \) are always less than \( i \) if any is present,
\[
\forall m'_1(\text{loop}_c^1) m'_2(\text{loop}_c^2) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{\ast} (S''_1, m'_1(\text{loop}_c^1)), \text{loop}_c^1(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{\ast} (S''_2, m'_2(\text{loop}_c^2)), \text{loop}_c^2(n_2) < i, S_1 \text{ and } S_2 \text{ terminate in the same way, produce the same output sequence, and have equivalent computation of output deciding variables in both } S_1 \text{ and } S_2 \text{ and the input sequence variable, } (S_1, m_1) \equiv_H (S_2, m_2) \text{ and } (S_1, m_1) \equiv_O (S_2, m_2) \text{ and } \forall x \in (OVar(S_1) \cap OVar(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2);
\]

2. The loop counter of \( S_1 \) and \( S_2 \) are of value less than or equal to \( i \), and there are no reachable configurations \( (S_1, m_1(\text{loop}_c^1, \sigma_1)) \) from \( (S_1, m_1(\sigma_1)) \), \( (S_2, m_2(\text{loop}_c^2, \sigma_2)) \) from \( (S_2, m_2(\sigma_2)) \) where all of the following hold:
• The loop counters of $S_1$ and $S_2$ are of value $i$, $\text{loop}_c^1(n_1) = \text{loop}_c^2(n_2) = i$.

• Value stores $\sigma_{1,i}$ and $\sigma_{2,i}$ agree on values of output deciding variables in both $S_1$ and $S_2$ as well as the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1,i}(x) = \sigma_{2,i}(x)$.

3. There are reachable configurations $(S_1, m_1(\text{loop}_c^1, \sigma_{1,i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^2, \sigma_{2,i}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:

• The loop counter of $S_1$ and $S_2$ are of value $i$, $\text{loop}_c^1(n_1) = \text{loop}_c^2(n_2) = i$.

• Value stores $\sigma_{1,i}$ and $\sigma_{2,i}$ agree on values of output deciding variables in both $S_1$ and $S_2$ including the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1,i}(x) = \sigma_{2,i}(x)$.

The proof is by induction on $i$. Refer to our technical report for details [88].

5.3 Proof Rule for Enumeration Type Extension

Enumeration types allow developers to list similar items. New code is usually accompanied with the introduction of new enumeration labels. Figure 5.5 shows an example of the update. The new enum label $o_2$ gives a new option for matching the value of the variable $a$, which introduce the new code $b := 3 + c$.

To show updates “enumeration type extension” to be backward compatible, we assume that values of enum variables, used in the If-predicate introducing the new code, are only from inputs that cannot be translated to new enum labels.

In order to have a general definition of the update class, we show a relation between two sequences of enumeration type definitions, called proper subset.
Definition 26. (Extension Relation of Enumeration Types) Let $EN_1, EN_2$ be two different sequences of enumeration type definitions. $EN_1$ is a subset of $EN_2$, written $EN_1 \subset EN_2$, iff one of the following holds:

1. $EN_1 = \text{"enum id \{el_1\}"}$, $EN_2 = \text{"enum id \{el_2\}"}$ where labels in type “enum id” in $EN_1$ are a subset of those in $EN_2$, $el_2 = el_1, el$ and $el \neq \emptyset$;

2. $EN_1, EN_2$ include more than one enumeration type definitions

$EN_1 = \text{"enum id \{el_1\}, EN_1\"}$, $EN_2 = \text{"enum id \{el_2\}, EN_2\"}$ where one of the following holds:

(a) $(EN_1' \subset EN_2')$ and $(el_1 = el_2) \lor (el_2 = el_1, el)$;

(b) $(EN_1' \subset EN_2') \lor (EN_1' = EN_2')$ and “enum id \{el_1\}" $\subset \text{"enum id \{el_2\}"}$.

Definition 27. (Enumeration Type Extension) Let $P_1, P_2$ be two programs where enumeration type definitions $EN_1$ in $P_1$ are a subset of $EN_2$ in $P_2$, $EN_1 \subset EN_2$ and $E$ are new enum labels in $P_2$. A statement sequence $S_2$ in a program $P_2$ includes updates of enumeration type extension compared with a statement sequence $S_1$ in $P_1$, written $S_2 \approx_E S_1$, iff one of the following holds:

1. $S_2 = \text{"If(id==l) then\{S_2'\} else\{S_2'\}"}$ and all of the following hold:
   
   - $l \in E$;
   
   - The variable id is not lvalue in an assignment statement, “id := e” $\notin P_2$;
   
   - $S_2' \approx_E S_1$;

2. $S_1 = \text{"If(e) then\{S_1'\} else\{S_1'\}"}$, $S_2 = \text{"If(e) then\{S_2'\} else\{S_2'\}"}$ where

   $(S_2' \approx_E S_1') \land (S_2' \approx_E S_1')$;

3. $S_1 = \text{"while(n_1)(e) \{S_1'\}"}$, $S_2 = \text{"while(n_2)(e) \{S_2'\}"}$ where $S_2' \approx_E S_1'$;
4. $S_1 \approx^S_O S_2$;

5. $S_1 = S'_1; s_1$ and $S_2 = S'_2; s_2$ where $(S'_2 \approx^S_E S'_1) \land (S'_2 \approx^S_H S'_1) \land (\forall x \in \text{Imp}(s_1, id_{IO}) \cup \text{Imp}(s_1, id_{IO})) : (S'_2 \approx^S_x S'_1) \land (s_2 \approx^S_E s_1)$.

We show that two programs terminate in the same way, produce the same output sequence, and have equivalent computation of variables defined in both of them in executions if there are updates of enumeration type extension between them.

**Lemma 5.3.1.** Let $S_1$ and $S_2$ be two statement sequences in programs $P_1$ and $P_2$ respectively where there are updates of enumeration type extensions in $S_2$ of $P_2$ compared with $S_1$ of $P_1$, $S_2 \approx^S E S_1$. If $S_1$ and $S_2$ start in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ such that both of the following hold:

- Value stores $\sigma_1$ and $\sigma_2$ agree on values of output deciding variables in both $S_1$ and $S_2$ including the input sequence variable and the I/O sequence variable, $\forall x \in (OVar(S_1) \cup OVar(S_2)) \cup \{id_I, id_{IO}\} : \sigma_1(x) = \sigma_2(x)$;

- No variables used in $S_2$ are of initial value of enum labels in $E$, $\forall x \in \text{Use}(S_2) : (\sigma_2(x) \notin E)$;

- No inputs are translated to any label in $E$ during the execution of $S_2$;

then $S_1$ and $S_2$ terminate in the same way, produce the same output sequence, and when $S_1$ and $S_2$ both terminate, they have equivalent computation of used variables and defined variables,

- $(S_1, m_1) \equiv_H (S_2, m_2)$;

- $(S_1, m_1) \equiv_O (S_2, m_2)$;

- $\forall x \in OVar(S_1) \cup OVar(S_2) : (S_1, m_1) \equiv_x (S_2, m_2)$;
By induction on the sum of the program size of \( S_1 \) and \( S_2 \), size\((S_1) + \text{size}(S_1)\). Refer to our technical report for details [88].

We show a auxiliary lemma telling that the two programs with updates of enumeration type extension have same set of used variables and the same set of defined variables.

**Lemma 5.3.2.** If there are updates of enumeration type extension in a statement sequence \( S_2 \) against a statement sequence \( S_1 \), \( S_2 \equiv_E S_1 \), then the output deciding variables in \( S_1 \) are a subset of those in \( S_2 \), \( \text{OVar}(S_1) \subseteq \text{OVar}(S_2) \).

*Proof.* By induction on the sum of the program size of \( S_1 \) and \( S_2 \). \( \square \)

**Lemma 5.3.3.** Let \( S_1 = \text{while}_{(n_1)}(e) \{S_1'\} \) and \( S_2 = \text{while}_{(n_2)}(e) \{S_2'\} \) be two loop statements in programs \( P_1 \) and \( P_2 \) respectively where all of the following hold:

1. Enumeration types \( EN_1 \) in \( P_1 \) are a proper subset of \( EN_2 \) in \( P_2 \), \( EN_1 \subset EN_2 \), such that there are a set of enum labels \( E \) only defined in \( P_2 \);
2. When started in states agreeing on values of output deciding variables in both \( S_1' \) and \( S_2' \) as well as the input sequence variable and the I/O sequence variable, initial values of used variables in \( S_2' \) are not enum labels in \( E \), and there are no inputs in \( S_2 \)'s execution translated into any label in \( E \), \( \forall x \in \text{OVar}(S_1') \cup \{id_I, id_{IO}\} \forall m_1(\sigma_1) m_2(\sigma_2) : \sigma_1(x) = \sigma_2(x) \), and \( S_1' \) and \( S_2' \) terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in \( S_1' \) and \( S_2' \) as well as the input sequence variable and the I/O sequence variable \( ((S_1', m_1) \equiv_H (S_2', m_2)) \land ((S_1', m_1) \equiv_O (S_2', m_2)) \land (\forall x \in \text{OVar}(S_1) \cup \text{OVar}(S_2) \cup \{id_I, id_{IO}\}) : (S_1', m_1) \equiv_x (S_2', m_2)) \);
If \( S_1 \) and \( S_2 \) start in states \( m_1(\text{loop}_c^1, \sigma_1), m_2(\text{loop}_c^2, \sigma_2) \), with loop counters of \( S_1 \) and \( S_2 \) not initialized (\( S_1, S_2 \) have not executed yet), value stores agree on values of output deciding variables in \( S_1 \) and \( S_2 \) as well as the input sequence variable, the I/O sequence variable, initial values of used variables in \( S_2 \) are not of values as enum labels in \( E \), no inputs are translated into enum labels in \( E \), then, for any positive integer \( i \), one of the following holds:

1. Loop counters for \( S_1 \) and \( S_2 \) are always less than \( i \) if any is present,
   \[
   \forall m'_1(\text{loop}_c^1) m'_2(\text{loop}_c^2) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{\quad} (S''_1, m'_1(\text{loop}_c^1)), \text{loop}_c^1(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{\quad} (S''_2, m'_2(\text{loop}_c^2)), \text{loop}_c^2(n_2) < i, S_1 \text{ and } S_2 \text{ terminate in the same way, produce the same output sequence, and have equivalent computation of output deciding variables in both } S_1 \text{ and } S_2 \text{ and the input sequence variable, the I/O sequence variable, } (S_1, m_1) \equiv_H (S_2, m_2) \text{ and } (S_1, m_1) \equiv_O (S_2, m_2) \text{ and } \forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2);
   \]

2. The loop counter of \( S_1 \) and \( S_2 \) are of value less than or equal to \( i \), and there are no reachable configurations \((S_1, m_1(\text{loop}_c^1, \sigma_1))\) from \((S_1, m_1(\sigma_1))\), \((S_2, m_2(\text{loop}_c^2, \sigma_2))\) from \((S_2, m_2(\sigma_2))\) where all of the following hold:
   - The loop counters of \( S_1 \) and \( S_2 \) are of value \( i \), \( \text{loop}_c^1(n_1) = \text{loop}_c^2(n_2) = i \).
   - Value stores \( \sigma_1 \), and \( \sigma_2 \), agree on values of output deciding variables in both \( S_1 \) and \( S_2 \) as well as the input sequence variable and the I/O sequence variable, \( \forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_1(x) = \sigma_2(x) \).

3. There are reachable configurations \((S_1, m_1(\text{loop}_c^1, \sigma_1))\) from \((S_1, m_1(\sigma_1))\), \((S_2, m_2(\text{loop}_c^2, \sigma_2))\) from \((S_2, m_2(\sigma_2))\) where all of the following hold:
   - The loop counter of \( S_1 \) and \( S_2 \) are of value \( i \), \( \text{loop}_c^1(n_1) = \text{loop}_c^2(n_2) = i \).
• Value stores $\sigma_1$ and $\sigma_2$, agree on values of output deciding variables in both $S_1$ and $S_2$ as well as the input sequence variable and the I/O sequence variable, $\forall x \in (OVar(S_1) \cap OVar(S_2)) \cup \{id_I, id_{IO}\} : \sigma_1(x) = \sigma_2(x)$.

The proof is by induction on $i$. Refer to our technical report for details [88].

5.4 Proof Rule for Variable Type Weakening

In programs, variable types are changed either to allow for larger ranges (weakening) or smaller ranges to save space (strengthening). For example, an integer variable might be changed to become a long variable to avoid integer overflow or a long variable might be changed to an integer variable because the larger range of long is not needed. Adding a new enumeration value can is also type strengthening. Increasing array size is another example of weakening. Allowing for type strengthening or weakening is essentially an assumption about the intent behind the update. The kinds of strengthening or weakening that should be allowed are application dependent and would need to be defined by the user in general. The type weakening considered is either changes of type Int to Long or increase of array size. These updates fix integer overflow or array index out of bound. In order to prove the update of variable type weakening to be backward compatible, we assume that there are no integer overflow and array index out of bound in execution of the old program and the updated program. In conclusion, the old program and the new program produce the same output sequence because the integer overflow and index out of bound errors fixed by the new program do not occur.

We formalize the update of variable type weakening, then we show that the updated program produce the same output sequence as the old program in executions if there are no integer overflow or index out of bound exceptions related to variables
with type changes. First, we define a relation between variable definitions showing
the type weakening.

**Definition 28. (Cases of Type Weakening)** We say there is type weakening from
a sequence of variable definitions \( V_1 \) to \( V_2 \), written \( V_1 \uparrow_{\tau} V_2 \), iff one of the following holds:

1. \( V_1 = \text{"Int id"}, V_2 = \text{"Long id"}; \)

2. \( V_1 = \tau id[n_2], V_2 = \tau id[n_1] \) where \( n_2 > n_1; \)

3. \( V_1 = V'_1, \tau_1 id_1[n_1], V_2 = V'_2, \tau_2 id_2[n_2] \) where \((V'_1 \uparrow_{\tau} V'_2) \land (\text{"\( \tau_1 id_1[n_1] \) \uparrow_{\tau} \text{"\( \tau_2 id_2[n_2] \)}}); \)

4. \( V_1 = V'_1, \tau_1 id_1[n_1], V_2 = V'_2, \tau_2 id_2[n_2] \) where \((V'_1 \uparrow_{\tau} V'_2) \land (\text{"\( \tau_1 id_1[n_1] \) \uparrow_{\tau} \text{"\( \tau_2 id_2[n_2] \)}}); \)

The following is the generalized definition of variable type weakening.

**Definition 29. (Variable Type Weakening)** We say that there are updates of
variable type weakening in the program \( P_2 = Pmpt; EN; V_2; S_{entry} \) compared with the
program \( P_1 = Pmpt; EN; V_1; S_{entry} \), written \( P_2 \approx_S P_1 \), iff \( V_1 \uparrow_{\tau} V_2 \).

We show that two programs terminate in the same way, produce the same output
sequence, and have equivalent computation of defined variables in both programs in
valid executions if there are updates of variable type weakening between them.

**Lemma 5.4.1.** Let \( P_1 = EN; V_1; S_{entry} \) and \( P_2 = EN; V_2; S_{entry} \) be two programs
where there are updates of variable type weakening, \( P_2 \approx_S P_1 \). If the programs \( P_1 \) and
\( P_2 \) start in states \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) such that both of the following hold:

- Value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of variables used in \( S_{entry} \) as well as
the input sequence variable, the I/O sequence variable, \( \forall x \in \text{Use}(S_{entry}) \cup \{id_I, id_{IO}\} : \sigma_1(x) = \sigma_2(x); \)

94
• There is no integer overflow or index out of bound exceptions related to variables of type change;

then $S_{\text{entry}}$ in the program $P_1$ and $P_2$ terminate in the same way, produce the same output sequence, and when $S_{\text{entry}}$ both terminate, they have equivalent computation of defined variables in $S_{\text{entry}}$ in both programs as well as the input sequence variable, the I/O sequence variable,

- $(S_{\text{entry}}, m_1) \equiv_H (S_{\text{entry}}, m_2);$  
- $(S_{\text{entry}}, m_1) \equiv_O (S_{\text{entry}}, m_2);$  
- $\forall x \in \text{Def}(S) \cup \{id_I, id_{IO}\}$ :  
  $(S_{\text{entry}}, m_1) \equiv_x (S_{\text{entry}}, m_2);$  

Because $S_{\text{entry}}$ are the exactly same in both programs $P_1$ and $P_2$, we omit the straightforward proof. Instead, we show that, if there is no array index out of bound and integer overflow in executions of the old program, then there is no array index out of bound or integer overflow in executions of updated program due to the increase of array index and change of type Int to Long.

The proof is straightforward because the statement sequence $S$ is same in programs $P_1$ and $P_2$. Refer to our technical report for details [88].

5.5 Proof Rule for Exit on Errors

Another bug fix is called “exit-on-error”, which causes the program to exit in observation of application-semantic-dependent errors. Figure 5.6 shows an example of exit-on-error update. In the example, the fixed bugs refer to the program semantic error that $a = 5$. Instead of using an “exit” statement, we rely on the crash from expression evaluations to formalize the update class. In order to prove the update
of exit-on-error to be backward compatible, we assume that there are no application
related errors in executions of the old program. Therefore, the two programs produce
the same output sequence because the extra check does not cause the new program’s execution to crash.

The following is the generalized definition of the update class “exit-on-error”.

**Definition 30. (Exit on Error)** We say a statement sequence $S_2$ includes updates of
exit-on-err from a statement sequence $S_1$, written $S_2 \approx_{\text{Exit}}^S S_1$, iff one of the following holds:

1. $S_2 = \text{"If}(e) \text{then} \{\text{skip}\} \text{else} \{\text{skip}\}; S_1$;

2. $S_1 = \text{"If}(e) \text{then} \{S'_1\} \text{else} \{S'_1\}$, $S_2 = \text{"If}(e) \text{then} \{S'_2\} \text{else} \{S'_2\}$ where both of the following hold
   - $S'_2 \approx_{\text{Exit}}^{S} S'_1$;
   - $S'_2 \approx_{\text{Exit}}^{S} S'_1$;

3. $S_1 = \text{"while} (n_1)(e) \{S'_1\}$, $S_2 = \text{"while} (n_2)(e) \{S'_2\}$ where
   $S'_2 \approx_{\text{Exit}}^{S} S'_1$;

4. $S_1 \approx_{O}^{S} S_2$;

5. $S_1 = S'_1; s_1$ and $S_2 = S'_2; s_2$ such that both of the following hold:
   - $S'_2 \approx_{\text{Exit}}^{S} S'_1$;
   - $S'_2 \approx_{H}^{S} S'_1$;
   - $\forall x \in \text{Imp}(s_1, id_{IO}) \cup \text{Imp}(s_1, id_{IO}) : S'_2 \approx_{x}^{S} S'_1$;
   - $s_2 \approx_{\text{Exit}}^{S} s_1$;

Though the bugfix in Definition 30 is not in rare execution in the first case, the definition shows the basic form of bugfix clearly.
We show that two programs terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in both programs in valid executions if there are updates of exit-on-error between them.

Lemma 5.5.1. Let $S_1$ and $S_2$ be two statement sequences respectively where there are updates of exit-on-error in $S_2$ against $S_1$, $S_2 \approx_{\text{Exit}}^S S_1$. If $S_1$ and $S_2$ start in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ such that both of the following hold:

- Value stores $\sigma_1$ and $\sigma_2$ agree on values of variables used in both $S_1$ and $S_2$ as well as the input sequence variable and the I/O sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I, id_{IO}\}: \sigma_1(x) = \sigma_2(x)$;
- There are no program semantic errors related to the extra check in the update of exit-on-error in the execution of $S_1$;

then $S_1$ and $S_2$ terminate in the same way, produce the same output sequence, and when $S_1$ and $S_2$ both terminate, they have equivalent computation of defined variables in both $S_1$ and $S_2$ as well as the input sequence variable and the I/O sequence variable,

- $(S_1, m_1) \equiv_H (S_2, m_2)$;
- $(S_1, m_1) \equiv_O (S_2, m_2)$;
- $\forall x \in (\text{Def}(S_1) \cap \text{Def}(S_2)) \cup \{id_I, id_{IO}\}$: $(S_1, m_1) \equiv_x (S_2, m_2)$;

The proof is by induction on the sum of the program size of $S_1$ and $S_2$, $\text{size}(S_1) + \text{size}(S_2)$. Refer to our technical report for details [88].

We list the auxiliary lemmas below. One lemma shows that, if there are updates of exit-on-error between two statement sequences, then there are same set of defined variables in the two statement sequences, and the used variables in the update program are the superset of those in the old program.
Lemma 5.5.2. Let $S_2$ be a statement sequence and $S_1$ where there are updates of exit-on-error, $S_2 \approx_{\text{Exit}}^S S_1$. Then output deciding variables in $S_1$ are a subset of those in $S_2$, $OVar(S_1) \subseteq OVar(S_2)$.

Proof. By induction on the sum of the program size of $S_1$ and $S_2$. Refer to our technical report for details [88].

Lemma 5.5.3. Let $S_1 = \text{while}_{(n_1)}(e) \{S'_1\}$ and $S_2 = \text{while}_{(n_2)}(e) \{S'_2\}$ be two loop statements where all of the following hold:

- the output deciding variables in $S'_1$ are a subset of those in $S'_2$, $OVar(S'_1) \subseteq OVar(S'_2) = OVar(S)$;

- When started in states $m'_1(\sigma'_1), m'_2(\sigma'_2)$ where
  
  - Value stores agree on values of output deciding variables in both $S'_1$ and $S'_2$ as well as the input sequence variable, and the I/O sequence variable, $\forall x \in OVar(S'_2) \cup \{id_I, id_{IO}\} \forall m'_1(\sigma'_1) m'_2(\sigma'_2) : \sigma'_1(x) = \sigma'_2(x)$;

  - There are no program semantic errors related to the extra check in the update of exit-on-error in executions of $S'_1$ and $S'_2$;

then $S'_1$ and $S'_2$ terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in $S'_1$ and $S'_2$ as well as the input sequence variable, and the I/O sequence variable, $((S'_1, m_1) \equiv_H (S'_2, m_2)) \land ((S'_1, m_1) \equiv_O (S'_2, m_2)) \land (\forall x \in OVar(S) \cup \{id_I, id_{IO}\} :$

$$(S'_1, m_1) \equiv_x (S'_2, m_2));$$

If $S_1$ and $S_2$ start in states $m_1(\text{loop}^1_c, \sigma_1), m_2(\text{loop}^2_c, \sigma_2)$ respectively, with loop counters of $S_1$ and $S_2$ not initialized ($S_1, S_2$ have not executed yet), value stores agree on
values of used variables in $S_1$ and $S_2$, and there are no program semantic errors related to the extra check in the update of exit-on-error, then, for any positive integer $i$, one of the following holds:

1. Loop counters for $S_1$ and $S_2$ are always less than $i$ if any is present,

$$\forall m'_1(\text{loop}^1_c) m'_2(\text{loop}^2_c) : (S_1, m_1(\text{loop}^1_c, \sigma_1)) \rightarrow (S''_1, m'_1(\text{loop}^1_c)), \text{loop}^1_c(n_1) < i, (S_2, m_2(\text{loop}^2_c, \sigma_2)) \rightarrow (S''_2, m'_2(\text{loop}^2_c)), \text{loop}^2_c(n_2) < i, S_1 \text{ and } S_2 \text{ terminate in the same way, produce the same output sequence, and have equivalent computation of output deciding variables in } S_1 \text{ and } S_2 \text{ and the input sequence variable, the I/O sequence variable, } (S_1, m_1) \equiv_H (S_2, m_2) \text{ and } (S_1, m_1) \equiv_O (S_2, m_2) \text{ and } \forall x \in (OVar(S_1) \cup OVar(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2);$$

2. The loop counter of $S_1$ and $S_2$ are of value less than or equal to $i$, and there are no reachable configurations $(S_1, m_1(\text{loop}^1_c, \sigma_1))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}^2_c, \sigma_2))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:

- The loop counters of $S_1$ and $S_2$ are of value $i$, loop$^1_c(n_1) = loop^2_c(n_2) = i$.
- Value stores $\sigma_1$, and $\sigma_2$, agree on values of output deciding variables in $S_1$ and $S_2$ as well as the input sequence variable, and the I/O sequence variable, $\forall x \in (OVar(S_1) \cup OVar(S_2)) \cup \{id_I, id_{IO}\} : \sigma_1(x) = \sigma_2(x)$.

3. There are reachable configurations $(S_1, m_1(\text{loop}^1_c, \sigma_1))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}^2_c, \sigma_2))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:

- The loop counter of $S_1$ and $S_2$ are of value $i$, loop$^1_c(n_1) = loop^2_c(n_2) = i$. 

99
• Value stores $\sigma_1$ and $\sigma_2$ agree on values of output deciding variables in $S_1$ and $S_2$ including the input sequence variable and the I/O sequence variable,

$$\forall x \in (OVar(S_1) \cup OVar(S_2)) \cup \{id_I, id_{IO}\} : \sigma_1(x) = \sigma_2(x).$$

The proof is by induction on $i$. Refer to our technical report for details [88].

5.6 Proof Rule for Improved Prompt Message

If the only differences between two programs are the constant messages that the user receives, we consider that the two programs to be equivalent. We realize that in general it is possible to introduce new semantics even by changing constant strings. An old version might have incorrectly labeled output: “median value = 5” instead of “average value = 5” for example. We rule out such possibilities because all non-constant values are guaranteed to be exactly same. In practice, outputs could be classified into prompt outputs and actual outputs. Prompt outputs are those asking clients for inputs, which are constants hardcoded in the output statement. Actual outputs are dynamic messages produced by evaluation of non-constant expression in execution. The changes of prompt outputs are equivalent only for interactions with human clients. In order to prove the update of improved prompt messages to be backward compatible, we assume that the different prompt outputs produced in executions of the old program and the updated program, due to the different constants in output statements, are equivalent. Because the old program and the new program are exactly same except some output statements with different constants as expression $e$, we could show two programs produce the “equivalent” output sequence under the assumption of equivalent prompt outputs.

We formalize the generalized update of improved prompt messages, then we show that the updated program produce the same I/O sequence as the old program in
executions without program semantic errors. The following is the definition of the update class of improved prompt messages.

**Definition 31. (Improved User Messages)** A program $P_2 = P_{\text{mpt}_2}; \text{EN}; V; S_{\text{entry}}$ includes updates of improved prompt messages compared with a program $P_1 = P_{\text{mpt}_1}; \text{EN}; V; S_{\text{entry}}$, written $P_2 \approx^{S}_{\text{Out}} P_1$, iff $P_{\text{mpt}_2} \neq P_{\text{mpt}_1}$.

We give the lemma that two programs terminate in the same way, produce the equivalent output sequence, and have equivalent computation of defined variables in both programs in valid executions if there are updates of improved prompt messages between them.

**Lemma 5.6.1.** Let $P_1 = P_{\text{mpt}_1}; \text{EN}; V; S_{\text{entry}}$ and $P_2 = P_{\text{mpt}_2}; \text{EN}; V; S_{\text{entry}}$ be two programs where there are updates of improved prompt messages in $P_2$ compared with $P_1$. If $S_1$ and $S_2$ start in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ such that both of the following hold:

- Value stores $\sigma_1$ and $\sigma_2$ agree on values of variables used in $S_{\text{entry}}$ in both programs as well as the input sequence variable, $\forall x \in \text{Use}(S_{\text{entry}}) \cup \{\text{id}_I\} : \sigma_1(x) = \sigma_2(x)$;

- Value stores $\sigma_1$ and $\sigma_2$ have “equivalent” I/O sequence, $\sigma_1(\text{id}_{\text{IO}}) \equiv \sigma_2(\text{id}_{\text{IO}})$;

- The different prompt outputs in the update of improved prompt messages are equivalent;

then $S_1$ and $S_2$ terminate in the same way, produce the equivalent output sequence, and when $S_1$ and $S_2$ both terminate, they have equivalent computation of defined variables in $S_{\text{entry}}$ in both programs as well as the input sequence variable, $S_{\text{entry}}$ in the two programs produce the equivalent I/O sequence variable,
• \((S_{\text{entry}}, m_1) \equiv_H (S_{\text{entry}}, m_2)\);

• \(\forall x \in (\text{Def}(S_1) \cap \text{Def}(S_2)) \cup \{\text{id}_I\} : (S_{\text{entry}}, m_1) \equiv_x (S_{\text{entry}}, m_2)\);

• The produced output sequences in executions of \(S_{\text{entry}}\) in both programs are “equivalent”, \(\sigma_1(\text{id}_{IO}) \equiv \sigma_2(\text{id}_{IO})\).

The difference between prompt types in \(P_1\) and \(P_2\) can be either addition/removal of labels as well as the change of the mapping of labels with constants. The proof is straightforward because programs \(P_1\) and \(P_2\) have the same entry statement sequence and we have the assumption that different prompt outputs due to the difference of the prompt type are equivalent.

By induction on the sum of the program size of \(S_1\) and \(S_2\), size\((S_1) + \text{size}(S_2)\). Refer to our technical report for details [88].

We list the auxiliary lemmas below. One lemma shows that, if there are updates of improved prompt messages between two statement sequences, then there are same set of defined variables and used variables in the two statement sequences. The second lemma shows that, if there are updates of improved prompt messages between two loop statements, then the two loop statement terminate in the same way, produce the equivalent output sequence, and have equivalent computation of defined variables in both the old and updated programs as well as the input sequence variable.

**Lemma 5.6.2.** Let \(S_2\) be a statement sequence and \(S_1\) where there are updates of “improved prompt messages”, \(S_2 \simeq_{\text{Out}} S_1\). Then used variables in \(S_2\) are the same of used variables in \(S_1\), \(\text{Use}(S_1) = \text{Use}(S_2)\), defined variables in \(S_2\) are the same as used variables in \(S_1\), \(\text{Def}(S_1) = \text{Def}(S_2)\).

**Proof.** By induction on the sum of the program size of \(S_1\) and \(S_2\). Refer to our technical report for details [88].
Lemma 5.6.3. Let $S_1 = \text{while}_{(n_1)}(e) \{ S'_1 \}$ and $S_2 = \text{while}_{(n_2)}(e) \{ S'_2 \}$ be two loop statements where all of the following hold:

- There are updates of improved prompt messages in $S'_2$ compared with $S'_1$, $S'_2 \approx_{S_{\text{out}}} S'_1$;
- $S'_1$ and $S'_2$ have same set of defined variables, 
  $\text{Def}(S'_1) = \text{Def}(S'_2) = \text{Def}(S)$;
- $S'_1$ and $S'_2$ have same set of used variables, $\text{Use}(S'_1) = \text{Use}(S'_2)$;
- When started in states $m'_1(\sigma'_1), m'_2(\sigma'_2)$ where
  - Value stores agree on values of used variables in both $S'_1$ and $S'_2$ as well as the input sequence variable, $\forall x \in \text{Use}(S'_1) \cup \{\text{id}_I\} \forall m'_1(\sigma'_1) m'_2(\sigma'_2)$:
    $\sigma'_1(x) = \sigma'_2(x)$;
  - Values of the I/O sequence variable in value stores $\sigma'_1, \sigma'_2$ are equivalent, $\sigma'_1(\text{id}_{\text{IO}}) \equiv \sigma'_2(\text{id}_{\text{IO}})$;

then $S'_1$ and $S'_2$ terminate in the same way, produce the “equivalent” output sequence, and have equivalent computation of defined variables in $S'_1$ and $S'_2$ as well as the input sequence variable, $((S'_1, m_1) \equiv_H (S'_2, m_2)) \land (\forall x \in \text{Def}(S) \cup \{\text{id}_I\}) : (S'_1, m_1) \equiv_x (S'_2, m_2))$.

If $S_1$ and $S_2$ start in states $m_1(\text{loop}_c^1, \sigma_1), m_2(\text{loop}_c^2, \sigma_2)$ respectively, with loop counters of $S_1$ and $S_2$ not initialized ($S_1, S_2$ have not executed yet), value stores agree on values of used variables in $S_1$ and $S_2$, and there are no program semantic errors, then, for any positive integer $i$, one of the following holds:

1. Loop counters for $S_1$ and $S_2$ are always less than $i$ if any is present,

$$\forall m'_1(\text{loop}_c^1) m'_2(\text{loop}_c^2) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{\star} (S''_1, m'_1(\text{loop}_c^1)), \text{loop}_c^1(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{\star} (S''_2, m'_2(\text{loop}_c^2)), \text{loop}_c^2(n_2) < i, S_1 \text{ and } S_2 \text{ terminate}.$$
in the same way, produce the equivalent output sequence, and have equivalent computation of defined variables in both $S_1$ and $S_2$ and the input sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $\forall x \in (\text{Def}(S_1) \cap \text{Def}(S_2)) \cup \{id_I\} : (S_1, m_1) \equiv_x (S_2, m_2); S_1$ and $S_2$ produce the “equivalent” I/O sequence;

2. The loop counter of $S_1$ and $S_2$ are of value less than or equal to $i$, and there are no reachable configurations $(S_1, m_1(\text{loop}^{1^i}_c, \sigma_{1i}))$ from $(S_1, m_1(\sigma_1)),$

$$(S_2, m_2(\text{loop}^{2^i}_c, \sigma_{2i}))$$

from $(S_2, m_2(\sigma_2))$ where all of the following hold:

- The loop counters of $S_1$ and $S_2$ are of value $i$, $\text{loop}^{1^i}_c(n_1) = \text{loop}^{2^i}_c(n_2) = i$.
- Value stores $\sigma_{1i}$ and $\sigma_{2i}$ agree on values of used variables in both $S_1$ and $S_2$ as well as the input sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I\} : \sigma_{1i}(x) = \sigma_{2i}(x)$.
- Values of the I/O sequence variable in value stores $\sigma_{1i}(id_{IO}) \equiv \sigma_{2i}(id_{IO})$;

3. There are reachable configurations $(S_1, m_1(\text{loop}^{1^i}_c, \sigma_{1i}))$ from $(S_1, m_1(\sigma_1)),$

$$(S_2, m_2(\text{loop}^{2^i}_c, \sigma_{2i}))$$

from $(S_2, m_2(\sigma_2))$ where all of the following hold:

- The loop counter of $S_1$ and $S_2$ are of value $i$, $\text{loop}^{1^i}_c(n_1) = \text{loop}^{2^i}_c(n_2) = i$.
- Value stores $\sigma_{1i}$ and $\sigma_{2i}$ agree on values of used variables in both $S_1$ and $S_2$ as well as the input sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I\} : \sigma_{1i}(x) = \sigma_{2i}(x)$.
- Values of the I/O sequence variable in value stores $\sigma_{1i}, \sigma_{2i}$ are equivalent, $\sigma_{1i}(id_{IO}) \equiv \sigma_{2i}(id_{IO})$;

The proof is by induction on $i$. Refer to our technical report for details [88].
5.7 Proof Rule for Missing Variable Initializations

A kind of bug fix we call *missing-initialization* includes variable initialization for those in the imported variables relative to the I/O sequence variable in the old program. Figure 5.7 shows an example of missing-initializations. The initialization $b := 2$ ensures the value used in “output $b + c$” is not to be undefined. In general, new variable initializations only affect rare buggy executions of the old program, where there are uses of undefined imported variables relative to the I/O sequence variable in the program. Because DSU is not starting in error state, we assume that, in the proof of backward compatibility, there are no uses of variables with undefined variables in executions of the old program.

The following is the definition of the update class “missing initializations”.

**Definition 32. (Missing Initializations)** A statement sequence $S_2$ includes updates of missing initializations compared with a statement sequence $S_1$, written $S_2 \approx_{\text{Init}}^S S_1$, iff $S_2 = S_{\text{Init}}; S_1$ where $S_{\text{Init}}$ is a sequence of assignment statements of form “lval := v” and $\text{Def}(S_{\text{Init}}) \subseteq \text{Imp}(S_1, \{\text{id}_{IO}\})$;

Though the bugfix in the update of missing initializations are not in rare execution in the first case in Definition 32, the definition shows the basic form of bugfix clearly.

We show that two statement sequences terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in both programs in valid executions if there are updates of missing initializations between them.

**Lemma 5.7.1.** Let $S_1$ and $S_2$ be two statement sequences respectively where there are updates of “missing initializations” in $S_2$ compared with $S_1$, $S_2 \approx_{\text{Init}}^S S_1$. If $S_1$ and $S_2$ start in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively such that both of the following hold:
• Value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of variables used in both \( S_1 \) and \( S_2 \) as well as the input sequence variable and the I/O sequence variable, \( \forall id \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_1(id) = \sigma_2(id) \);

• defined variables in \( S_{\text{Init}} \) are of undefined value in value stores \( \sigma_1, \sigma_2 \), \( \forall id \in \text{Def}(S_{\text{Init}}) : \sigma_1(id) = \sigma_2(id) = \text{Udf}[\tau] \) where \( \tau \) is the type of the variable \( id \);

• There are no use of variables with undefined values in the execution of \( S_1 \);

• There are no crash in execution of \( S_{\text{Init}} \);

then \( S_1 \) and \( S_2 \) terminate in the same way, produce the same output sequence, and when \( S_1 \) and \( S_2 \) both terminate, they have equivalent computation of used variables and defined variables in both \( S_1 \) and \( S_2 \) as well as the input sequence variable and the I/O sequence variable,

• \( (S_1, m_1) \equiv_H (S_2, m_2) \);

• \( (S_1, m_1) \equiv_O (S_2, m_2) \);

• \( \forall x \in (\text{Def}(S_1) \cup \text{Def}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2) \);

By induction on the sum of the program size of \( S_1 \) and \( S_2 \), \( \text{size}(S_1) + \text{size}(S_2) \).

Refer to our technical report for details [88].
Chapter 6

SITBACK: THE STATE MAPPING TOOL

This chapter describes the implementation of our state mapping tool SitBack. The description includes our methodology of automatic state mapping, the architecture of SitBack, implementation limitations, and the evaluation of SitBack.

This chapter is organized as follows. We use one example to illustrate how we approach the DSU automatic state mapping problem for real world programs in Section 6.1. Section 6.2 presents a brief statement of the automatic state mapping problem for real world programs. The summary of our methodology for calculating state mapping is shown in Section 6.3. Section 6.4 introduces the architecture of SitBack as well as the details of each component of SitBack. The limitations of SitBack is discussed in Section 6.5. Section 6.6 includes the detailed evaluation of SitBack using real world program updates.

6.1 Motivating Example

Solving the state mapping problem requires an understanding of the semantic differences and similarities between two programs, the semantic equivalence problem, which is undecidable by Rice’s Theorem [89]. However, the semantic equivalence problem is solvable in many cases of practical interest by considering a strong form of equivalence that essentially requires step-by-step execution equivalence. The semantic equivalence problem has been studied by researchers interested in determining how different versions of a program are related and how those versions could be merged together [53]. The state mapping problem that we address is related to, but different
from, the semantic equivalence problem. Figure 6.1 gives an example of an updated program fragment to illustrate some of the differences.

First we note that the two fragments cannot be equivalent under a strict definition of equivalence because the variables have different types. This requires the introduction of type compatibility rules that allow us to state when a new declaration is compatible with the old declaration. Furthermore, in the old fragment, the fourth statement is always executed whereas in the updated code, the execution is predicated on the conditional variable \( E \). If \( E \) is true, statement 6’ which is equivalent to statement 6 is executed, otherwise a new statement (8’) is executed. Finally, the
enumerated type \textit{opt} is extended with a new value $0_3$ and a new statement (10') is executed if $e = 0_3$.

If we ignore type differences and check for semantic equivalence as done in [53] between the statements of the two fragments, we can determine that 3 is equivalent to 4' and 4 is equivalent to 3' (reordered statements with no dependencies) with no other statements being equivalent. Our aim is to determine backward compatibility and not semantic equivalence. We want to map the state of the execution of the old code fragment to a valid state of execution of the new fragment. The new code fragment can be thought of as extending the functionality of the old fragment. The variable $E$ is a new boolean variable; if it is \texttt{true} then the old functionality is provided, otherwise the new functionality is provided; the old code fragment can be thought of as implicitly assuming that $E$ is \texttt{true}. The new statement whose execution is predicated on the value of $e$ is a new option that was not available in the old fragment. If $e$ is used as an enumerated type (as intended) and not manipulated as an integer, then we can assume that the new statements (9' and 10') could have been included in the old fragment without statement 10' being executed because $e$ should not take the value $0_3$ in the old fragment (as a user-provided value for example). A state mapping between the two fragments would map $(a, b, c, d, e, \texttt{true}) \mapsto (a', b', c', d', e', E)$. This mapping guarantees that once the new state is loaded, the \textit{resulting execution is an extension of a valid execution of the new program}, ignoring type differences for now. \textit{This is how we judge the correctness of a dynamic update}. We assume that a statically updated program is correct and we aim to ensure that the dynamically updated program provides \textit{the same semantics}; in other words we use the new program as its own specification.
We also require, in a sense that we make precise in the chapter, which a subset of the executions of the new program corresponds to executions of the old program.

The example we just presented shows the two ways we handle changes to the applications. Some changes are ignored (such as some type changes and the new enumeration value) because we can reason that the execution of the application before an update occurs is consistent with these changes. Other changes cannot be ignored and the state mapping needs to be calculated in such a way that the new application is backward compatible with the old one. This category includes initializing configuration variables (such as the variable \( E \) above) which is a special case of code generalization (adding a parameter to a function to make it more flexible). It is important to emphasize that our mapping works under the assumption of backward compatibility.

For concreteness, consider a web server that allows anonymous login by default in an old version and controls anonymous login with a variable \( \text{anon} \) that is set at configuration time in the new version (similar to the variable \( E \) above). For backward compatibility, we would map \( \text{anon} \) to \texttt{true}. It is possible though that the system administrator would want to disallow anonymous login after installing the new version. In such a case, \( \text{anon} \) needs to be mapped to \texttt{false}, but this is not something that a tool can handle in general. In particular, what should be done to anonymous connections at the time of the update? Under the assumption of backward compatibility, we would leave them unchanged. Under a change of policy, such connections might be allowed to proceed while prohibiting new anonymous connections (this would violate the semantics of the application under the new system configuration) or such anonymous connections need to be terminated. Our tool can in principle (and non-trivially) be

\footnote{Note that this validity requirement is stronger than that suggested by Gupta [42] in which an update is considered valid if eventually a valid state of the new application is reached.}
extended to support such policy changes with the help of user annotations, but that is beyond the scope of this chapter; we only provide support for backward compatibility by default.

6.2 Automatic Calculation of State Mapping for Real World Programs

Currently state mapping for DSU is mostly done by developers manually. There are several issues of manual state mapping. First, it is unclear what principle is adopted by developers to create the state mapping. Consequently, it is unclear what properties of consequent DSU are. Second, manual state mapping is essentially not sound because there is no rigid process of generating state mapping manually. The process of calculating a manual state mapping is different from a developer to another. Third, it is unclear what knowledge of programs is required to come up with state mapping. In presence of significant amount of updates, developers are tempted to minimize the amount of state mapping by restricting update points; it is unclear how developers manage to decide which update points to choose to minimize state mapping workload. Hence manual state mapping is not sound from developers’ own reasoning. To address issues of manual state mapping, we intend to automate state mapping for backward compatible DSU because this helps improve DSU correctness.

In this chapter, we show how to automate state mapping for real world programs to achieve backward compatibility. We developed a prototype tool to automate the construction of state mappings for real world programs. Our developing of automatic state mapping helps explore the practical challenges due to complexities from real world programs (e.g., procedure, jump instruction, alias). Our state mapping tool is an extension to UpStare [67]. Due to this fact, the state mapping tool is only applicable for C programs.
The backward compatibility is a relation between an old program and a new one. To achieve backward compatibility, we need to relate the old program with the new one. Due to the complexity in real world programs such as procedure, we could not use syntactic conditions of backward compatibility presented in prior chapters. It is required to find a different way of relating programs for backward compatibility.

We summarize the challenges of automatic state mapping for C programs.

1. Challenges due to program organization.

   (a) A real world program is usually composed of procedures. Furthermore, there are recursive procedures which add to the complexity of relating two programs. Our syntactic conditions of backward compatibility do not apply for real world programs immediately.

   (b) A real world program is usually composed of several parts. Except the main part that does the required job, there is usually a part of the program that is not directly related to the job, namely, the logging. The logged data could be recorded either by dumping to a file or sending to a remote party. In real world programs, the logging has different ways of implementation. It is desirable to identify the logging from the main part of the program.

2. Challenges due to programming language features. C language has features that facilitate developers to create versatile programs, for example, recursion, variable pointer, jump instruction, dynamic struct type like linked list, to name a few. These features adds to the complexity for program analysis [94, 87]. For instance, it is still an open question as to how to accurately determine a data structure on dynamically allocated storage [87].
6.3 Methodology of Generating State Mapping

Our algorithm for generating state mappings for backward compatible programs uses program dependence graphs (PDG) [52]. A full definition of PDG is omitted. We use the following brief description of PDGs taken from [52]: The PDG for a program \( P \), denoted by \( G_P \), is a directed graph whose vertices are connected by several kinds of edges. The vertices in \( G_P \) represent the assignment statements and predicates of \( P \) (we also allow vertices to include function calls). In addition, \( G_P \) includes a special \textit{Entry} vertex, and also includes one \textit{Initial definition} vertex for every variable \( x \) that may be used before being defined. The edges of \( G_P \) represent control and data dependence. For programs with no \texttt{goto} statements, control dependence edges reflect the nesting structure of the program. Data dependence edges include both \textit{flow} dependence edges and \textit{def-order} (definition order) dependence edges. Flow dependence edges represent possible flow of values. Def-order dependence edges are included in a PDG to ensure that inequivalent programs cannot have isomorphic PDGs.

It is a well known result that two programs are strongly equivalent if they have isomorphic PDGs [52]. This result is restricted to programs with structured control flow and has been extended to programs with multiple procedures using System Dependence Graphs (SDG) [21] and to programs with variables from dynamically allocated storage [82]. The result is not applied programs with goto statements though there is solution to do PDG-based program slicing for programs with jump statements [17]. For such programs, we have our PDG extension to ensure that isomorphic PDGs imply equivalent programs. We describe our PDG extension in Section 6.4.

At a high level, our state mapping generation algorithm attempts to independently match the PDGs of individual functions. Attempting to match two functions
introduces constraints that are used in calculating the state mapping for individual functions and the whole program. The algorithm might not succeed in matching all functions, in which case some functions need to be matched by the user who provides a state mapping or provides annotation to help the algorithm in matching additional functions. In what follows we describe how individual matching and whole program matching are done.

6.3.1 Matching Individual Functions

To calculate a state mapping for two functions, we consider the PDGs of the old and new version, $G_{old}$ and $G_{new}$, and try to determine if the two graphs can be made to be isomorphic. Nodes that correspond in an isomorphism are called matched nodes. Unlike strict equivalence, we allow the removal of some nodes or assigning initial values to some (new) variables to find an isomorphism. Nodes that correspond to log functions, for example, can be removed from both PDGs. Nodes that correspond to exit functions are considered equivalent. Conditional branching nodes in $G_{new}$ can be eliminated by setting the condition variable to either true or false if we determine that the variable is a new variable whose value does not depend on existing variables. Conditional nodes that check if a variable is equal to a newly introduced enumeration value are also eliminated as the newly introduced values cannot occur in the old program (in non-error executions). Matching two functions is driven by $G_{old}$. At the end of the matching, all nodes of $G_{old}$ must be matched. If all the nodes of $G_{old}$ are matched to nodes of $G_{new}$, any remaining nodes non-matched nodes in $G_{new}$ need to be either ignored (bug fixes for example) or that they are part of separation of state; otherwise, the matching is not successful. In the following sections we explain how PDG nodes are matched, and then we explain how generalized PDG matching is done.
using a backtracking algorithm. Finally, we explain how whole program matching is done.

**Matching Nodes**

As it executes, the algorithm maintains a list of matched variables. These are variables from the new program that can match variables from the old program. This list might change due to some conflict with matching done in a different part of the PDGs. In the algorithm, the matched variables are maintained in a stack, `matched_pairs`. Elements of the stack contain the matched variables and the corresponding PDG nodes that contributed to matching them.

A basic step that is done repeatedly in the algorithm is the attempt at matching two nodes \( n_1 \) and \( n_2 \) from the two PDGs. We explain how that is done first (this is also done in works for semantic equivalence checking), but we do not provide corresponding pseudo code.

1. Determine if \( n_1 \) and \( n_2 \) have the same operator by reducing them to a canonical form. The operator of a node captures the arithmetic or control flow done at that node. For example, the operator of both \( a = b \cdot c + 2 \) and \( a' = b' \cdot c' + 2 \) is \( v_1 = (v_2 \cdot v_3) + 2 \). Operators are not restricted to arithmetic operators. If nodes do not have the same operator, return ERROR.

2. If \( n_1 \) and \( n_2 \) have the same operator, check if the variable matching implied by the operator is consistent. For example, \( a = a + b \) and \( a = b + c \) have the same operator but are not consistent because \( a \) is mapped to \( a \) and to \( b \). If the matching is not consistent, return ERROR. However, we allow difference for constant operands in an array index. For example, \( a = b[c + 1] \) and \( a = b[c + 2] \)
match because we find most changes of constant operands are due to bug fixes for errors like index out of range.

3. If the matching found in step 2 is not consistent with the existing matching on the stack, return ERROR.

4. Check if the dependence edges of \( n_1 \) and \( n_2 \) are consistent. This is to ensure that two PDGs are isomorphic. All kinds of edges in PDG are checked including incoming/outgoing flow edges, incoming/outgoing/witness definition order edges. Control edges are implicitly checked by the matching algorithm. In checking dependence edges, we only consider edges that already have one endpoint (source or target) already matched. For edges with nodes that are not already matched we assume that they will eventually match or that the nodes will disappear due to some node elimination or new nodes will appear due to restoration of the PDG. This is one reason we do not use the number of the edges of various types as a preliminary signature for matching nodes.

**Generalizing PDG Matching**

The generalized PDG matching deals with generalization in the form of new conditions after trimming PZDGs for negligible code. As a first step, all code that can be ignored is removed from the matched functions and the PDGs are accordingly updated. This requires considerable analysis as described in the semantic analysis in the chapter. This leaves two PDGs that need to be matched with the PDG of the updated function possibly having new conditions, variable renaming, statement reordering, and new state that can be separated (as described above). We give an informal description of the algorithm. The algorithm is a brute force backtracking algorithm that attempts to match nodes by matching the nodes individually then matching their children.
depth first. If two nodes cannot match, then an attempt is made to eliminate the new node if it is a new condition node. Alias analysis is done to ensure that the new condition is indeed a new condition. true is tried first and if that fails and the matching backtracks to the same node, then false is attempted. If both true and false fail, then an attempt is made to consider the children out of order (expensive). If all out of order fail, then the algorithm backtracks. If the algorithm backtracks to the entry nodes, then there is no matching. If the algorithm succeeds in finding a matching, then the values of new condition are used in constructing the mapping. The details of our backtracking algorithm is shown in Section 6.4.

6.3.2 Matching Whole Programs

We do not use SDGs explicitly to solve the state mapping for whole programs. Instead we process functions separately by finding state mappings for functions with the same names (We also have heuristics for dealing with renamed functions by checking the “edit distance” between unmatched function bodies, but we omit that from the description because the effectiveness of our heuristic is not confirmed from our experiments). Every one of these mappings might introduce constraints (assumptions) on some global variables (by setting their values to true or false for example). When mapping two functions, any function calls are assumed to match if the functions have the same names and their parameters are matched variables. If one parameter is a constant in the old version and a variable in the new version, then an assumption is recorded that the variable is equal to the constant and the rest of the matching is done under that assumption. If at the end of the procedure, all functions are mapped and their assumptions do not conflict, then putting the mappings together will give a global mapping. Matching whole programs is further described in the next section.
In overview, our state mapping is relying on the use of program dependence graph [52]. The architecture of state mapping tool SitBack is shown in Figure 6.2.

A program is at first annotated instructing SitBack which part of difference could be ignored in state mapping. Then the program source is merged into one file by UpStare using CIL [76]. Then each of the merged programs is fed to alias analyzer to produce context sensitive, flow insensitive alias information. We next feed annotated programs and their alias information to the mapping generator. The mapping generator calls semantic analyzer to identify logging and exit functions in both programs separately using alias information. With identified logging and exit functions, mapping generator does PDG matching for updated procedures in the topology order of call graph of these changed procedures. Finally, the mapping generator output the result of PDG matching.
We next explain how we perform alias analysis, identify logging and exit procedures, create PDG/slice, do PDG matching and generate patch in order.

### 6.4.1 Alias Analysis

Our alias analysis is a standard context sensitive and flow insensitive analysis, but we combine it with value tracking to track the values that file descriptors can take. In addition to the points-to set typically maintained by alias analysis, we maintain a value set which contains file names. Tracking file descriptors is needed for the detection of log files, which are write-only files as we already described. Alias analysis allows us to determine for each read() or write() the files that are accessed by the calls. These would be the files in the value set of the file descriptor used by the call. Heap objects are treated by assuming that all objects of the same type are aliases of each other. This is achieved by associating with each field struct t a points-to set and a value set. These sets are associated with the type t and not individual objects of that type. They are updated with every assignment to a field of an object of type t.

Algorithm 1 shows pseudo code for alias analysis combined with file descriptor tracking. Each global or local variable x (or field of a structure) is assigned two sets v(x) and p(x). v(x) is the set of values that x can take and p(x) is the set of locations x can point to. Initially, these sets are empty. Initialization is not shown in the pseudo code. The analysis is a fixed-point flow insensitive calculation and goes over the list of all statements of all functions until there is no change. Of particular interest are system calls that open a file (or wrappers to such calls). This is the base case for adding a value to the set v(x).

Simple assignments without following pointers propagate value and pointer sets. The assignment x = &y is the base case for adding elements to p(x); it adds y to p(x).
Algorithm 1 Alias Analysis Algorithm

1: repeat
2:   for all function f() in program do
3:     for all statement s of f() do
4:       process(s)
5:     end for
6:   end for
7: until no change
8: process(s)
9: switch s do
10:   case x = fopen("filename") :
11:     v(x) = v(x) ∪ "filename"
12:   case x = &y :
13:     p(x) = p(x) ∪ {y}
14:   case *x = &y :
15:     for all z ∈ p(x) do
16:       p(z) = p(z) ∪ {y}
17:     end for
18:   case x = y :
19:     v(x) = v(x) ∪ v(y)
20:     if scal or ptr(x) then
21:       p(x) = p(x) ∪ p(y)
22:     end if
23:   case x = malloc() :
24:     p(x) = p(x) ∪ {HEAP}
25:     for all z such that HEAP ∈ p(z) do
26:       if type(z) == type(x) == struct t then
27:         v(t) = v(t) ∪ v(z)
28:       end if
29:     end for
30:   case "x = non_malloc()" :
31:     v(x) = v(non_malloc)
32:     if scal or ptr(x) then
33:       p(x) = p(non_malloc)
34:     end if
35:   case x = q → y or (*q).y :
36:     for all z ∈ p(q) do
37:       process(x = z.y)
38:     end for
39:   case *x = *y :
40:     for all z, w ∈ p(y) do
41:       v(z) = v(z) ∪ v(w)
42:       if scal or ptr(z) then
43:         p(z) = p(z) ∪ p(w)
44:       end if
45:     end for
46:   case x = f(a₁; a₂; ...; aₖ) :
47:     Formal params are x₁, x₂, ..., xₖ;
48:     if f is a function pointer then
49:       for all z ∈ p(f), i=1..k do
50:         process(xᵢ = aᵢ)
51:       end for
52:     else
53:       for all i=1..k do
54:         process(tᵢ = aᵢ)
55:       end for
56:     end if
57:   case return x :
58:     let f be the function in which "return x" appears;
59:     if f is a function pointer then
60:       for all z ∈ p(f) do
61:         process(x = z)
62:       end for
63:     else
64:       for all i=1..k do
65:         process(tᵢ = aᵢ)
66:       end for
67:     end if
68:   end case
69: end switch
70: for all field fd in struct t do
71:   process(x.fd = w.fd)
72: end for
73: end repeat
The assignment \( *x = \&y \) is handled similarly, but \( \{y\} \) is added to \( p(z) \) for all \( z \in p(x) \) because an assignment to \( *x \) affects all locations that \( x \) points to. Four cases are considered depending on whether none, one, or both sides are pointer dereferences. If the left side of an assignment is \( *x \), then the values of all locations pointed to by \( x \) are updated by the assignment. If the right side of an assignment is \( *y \), then the values of all locations pointed to by \( y \) are used in updating the value of \( x \). For the case of structures, assignments affect individual fields. We only show two cases of assignments without following pointers. Assignments in which one of the sides is of the form \( p \rightarrow y \) or \( (*p).y \) are handled similarly. There are a number of cases depending on whether one or both sides are of the form \( p \rightarrow y \) or \( (*p).y \). Also, we need to consider cases in which those forms are themselves dereferenced as in \( *(p \rightarrow y) \). There are six such cases in total. We only show one such case in the pseudo code.

Interprocedural propagation of values and pointer sets is handled when a statement includes a function call. A function call is treated as a sequence of assignments from an actual parameter \( a_i \) to a formal parameter \( x_i \). Function pointers are tracked (incompletely shown) and, therefore, interprocedural propagation is also done for the case of function pointers. A call by using a function pointer \( f \) is equivalent to a set of calls for every function in the points-to set of that function pointer \( p(f) \). For each function we associate one variable (the function variable) that keeps track of the values that can be returned by the function. This points-to and value sets of the variable are updated when return statements are processed; they are treated as assignment to the function variable. For example, in a function call \( x = f(...) \), \( v(x) \) is then the value set of the function.

The pseudo code only shows a base case with \( x_i \) and \( a_i \) for the parameters. But there are more cases similar to assignment statement depending on whether none, one, or both \( x_i \) and \( a_i \) are dereferences or if \( a_i \) is of the form \( \&y \). We also omit
the case where the function call is the right-hand side of an assignment. We do not consider function call as part of expressions because CIL already separates such calls into individual assignments.

For a non-malloc library function or calculation “y op z”, v(x) is then ALL, which indicates that any value can be taken by the variable; if x is a pointer then p(x) is also ALL because a calculated value can point anywhere.

6.4.2 Semantic Analyzer

The semantic analyzer in SitBack is mainly to identify logging and exit functions, which are used to identify the part of a program where the change does not affect the program semantics. We do not care changes of logging. Either we do not care changes of how a program exits.

Based on our study of real world programs (e.g. Vsftpd, Sshd, Icecast), we identify common characteristics of logging and exit functions. Then we design heuristics to identify logging and exit functions in programs. We describe how we identify logging and exit functions below in order.

From our study of programs, we found that the log file is only written but not consumed by any means. So the first step is to identify files that are only written in the program. Because any program analysis does not know which system routines are used to read or write a file, we require that developers provide the list of library routines reading and writing a file. We next look at alias information of file descriptors at the location of file reading and writing. By alias information, we could identify the log file and basic logging functions. From our observation, we found that a real world program usually includes wrappers of logging function. We recognize a wrapper of a logging function to be a caller of the logging function and nothing else. Our heuristic
may miss a complicated wrapper. However, we find that most wrappers are only calling a basic logging function or another wrapper.

The identification of exit functions is similar to that of logging functions. We require that developers specify exit library routines. An exit function should not include any statements that may take indefinite amount of time to finish such as loop or recursion or waiting operation. We also check wrappers of exit functions.

6.4.3 PDG/Slice Generation

We first explain how we create program dependence graph, then we show how we create PDG slice.

**Algorithm 2** PDG Generation Algorithm

1: Input : the function (with annotation), logging functions, exit functions
2: Create control dependence edges;
3: Use reaching definitions to create flow edges;
4: Use flow edges to create definition order edges;
5: Eliminate PDG nodes for logging function calls;
6: Eliminate nodes corresponding to annotated lines;
7: Replace any PDG node for an exit function by a uniform exit node;
8: Connect gotos with destination nodes;
9: Create anti-control edges;

Algorithm 2 shows how we create PDG. We use CIL to traversal a function in a syntax-directed way, which enables the creation of control dependence edges.

We use reaching definitions by CIL to create flow edges, loop-independent ones and loop-carried ones. The reaching definition by CIL does not work for fields in a struct variable. We go around the problem by struct variable flattening. The
struct variable flattening is to consistently replace the defined/used structure fields by manually created variables before the calculation of reaching definitions. For example, a statement “x.f = expr” is replaced by “stbk-x-f = expr” where “stbk-x-f” is a manually created variable. Based on the flow edges, we could get definition order edges based on its definition in [52].

We next trim PDG to remove logging functions and annotation. We then replace any exit function by a uniform “exit” because we do not care how a program exits.

We last handle the control flow of arbitrary jump (goto) by connecting a goto statement with any skipped statement. A skipped statement by a goto statement $s_{go}$ is a statement that is dominated by the goto statement $s_{go}$ and is post dominated by the label that the goto $s_{go}$ jumps to. Figure 6.3 shows an example of anti control edges. The dotted edges are anti control edges, the two-dots dashed edge is the control edge from goto statement to the goto label.

In a real world program, there can be either forward jump (to a later program point) or backward jump (to a prior program point). It is not clear how to extend PDG for programs with arbitrary forward/backward jumps in order to check program equivalence from PDG isomorphism. Fortunately, in the studied real world program evolution, we only notice forward jumps that jump to a program point around the

```
1: if(b != 0 & & a % b == b % a)
2: goto earlystop;
3: b = b * 2 + c;
4: d = c * 2 << b < a;
5: if(C)
6: a <<= 1;
7: else
8: a >>= 1;
9: earlystop;
10: exit(1);
```

Figure 6.3: An Example of Anti-Control Edges
end of a function that post-dominates the entry of the function. The particular type of forward jumps is not a concern because we can imagine equivalently transforming a forward jump of the particular type by adding conditional over the statements skipped by the jump till the target program point of the jump. The transformation includes extending the expression of any nested loop predicate of the goto statement. For example, “s1;s2;if(e) {goto L};s3;L:s4” is equivalent to “s1;s2;if(e) {flag = 1;} ;if(!flag) s3;L:s4”. Then our PDG extension of the jump statement captures the equivalent transformation of a jump statement by adding anti-control dependence from a goto statement to any skipped statement by the goto. Fig 6.4 shows the statistics of changed functions with at least one forward jump of the particular type described above. From the case study of vsftpd evolution, we see that there are usually a few functions in the vsftpd program that include the particular type of forward jumps and are frequently updated.

Our creation of PDG slice is to iteratively keep the control and data dependence from the interested PDG node and variable.

6.4.4 PDG Matching

The overview of matching two PDGs [52] is shown in Algorithm 3. The PDG matching is roughly of two steps. We first do the PDG matching using backtracking, which is shown in algorithm 4, 5 and 6; the handled updates include specializing new configuration variables, type weakening, prompt message improvement and function generalization. Then, we justify the unmatched part in PDG2 for updates like separation of states, enum type extension, exit-on-error and missing initialization; this step is shown in algorithm 7.

We went through our PDG matching algorithm in Algorithm 4. In lines 8 and 13 of algorithm 4, we match nodes in PDG1 and PDG2 in order. The order of
<table>
<thead>
<tr>
<th>Program</th>
<th>Update</th>
<th>Changed Functions with At Least One Forward Jump of the Particular Type</th>
<th>Number of Changed Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vsftpd</td>
<td>1.1.0 - 1.1.1</td>
<td>handle_upload_common</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>1.1.1 - 1.1.2</td>
<td>N/A</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>1.1.2 - 1.1.3</td>
<td>N/A</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>1.1.3 - 1.2.0</td>
<td>N/A</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>1.2.0 - 1.2.1</td>
<td>vsf_filename_passes_filter, handle_dir_common, handle_retr, handle_upload_common</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>1.2.1 - 1.2.2</td>
<td>N/A</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>1.2.2 - 2.0.0</td>
<td>handle_retr, handle_upload_common</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>2.0.0 - 2.0.1</td>
<td>N/A</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2.0.1 - 2.0.2</td>
<td>handle_retr, handle_dir_common, handle_upload_common</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>2.0.2 - 2.0.3</td>
<td>handle_eprt</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>2.0.3 - 2.0.4</td>
<td>handle_upload_common, handle_retr, vsf_filename_passes_filter</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>2.0.4 - 2.0.5</td>
<td>handle_retr, handle_upload_common</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>2.0.5 - 2.0.6</td>
<td>handle_retr, handle_upload_common</td>
<td>20</td>
</tr>
<tr>
<td>Sshd</td>
<td>3.6.1p1 - p2</td>
<td>N/A</td>
<td>16</td>
</tr>
<tr>
<td>Cherokee</td>
<td>0.4.2.b1 - b2</td>
<td>cherokee_handler_redir_init</td>
<td>12</td>
</tr>
<tr>
<td>Less</td>
<td>335 - 337</td>
<td>N/A</td>
<td>2</td>
</tr>
<tr>
<td>Pexec</td>
<td>1.0rc7 - rc8</td>
<td>N/A</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 6.4: Statistics of Updated Functions with At Least One Forward Jump of the Particular Type
Algorithm 3 CompletePDGMatch(PDG1, PDG2, L\textsubscript{NV}, L\textsubscript{EN}, L\textsubscript{EX}, L\textsubscript{VM}, L\textsubscript{X})

1: Input: PDG1, PDG2

2: Input: L\textsubscript{NV}, list of (newConfigVar) and (newFormalParam)

3: Input: L\textsubscript{EN}, list of (extEnumLabel)

4: Input: L\textsubscript{EX}, list of (exitFuncName) in PDG2

5: Input & Output: L\textsubscript{VM}, list of (varInProg1, varMatchInProg2)

6: Input & Output: L\textsubscript{X}, list of (newConfigVar $\rightarrow$ assumVal) and (newFormalVar $\rightarrow$ assumVal)

7: Output: Decision whether the matching of PDG1 and PDG2 is successful

8: Output: L\textsubscript{M}, list of (nodeInG1, nodeMatchInG2)

9:

10: (Res, L\textsubscript{M}, L'\textsubscript{VM}, L'\textsubscript{X}) := TryMatchPDG(PDG1, PDG2, L\textsubscript{NV}, L\textsubscript{VM}, L\textsubscript{X});

11: if failure = Res then

12: return (failure, $\emptyset$, $\emptyset$, $\emptyset$);

13: end if

14: if isExactMatch(PDG1, PDG2, L\textsubscript{M}) then

15: return (success, L\textsubscript{M}, L'\textsubscript{VM}, L'\textsubscript{X});

16: else if (isPDG1MatchPartOfPDG2(PDG1, PDG2, L\textsubscript{M})) and

(AllJustified = JustifyUnmatchedInPDG2(PDG2, L\textsubscript{EN}, L\textsubscript{EX}, L\textsubscript{M})) then

17: return (success, L\textsubscript{M}, L'\textsubscript{VM}, L'\textsubscript{X});

18: else

19: return (failure, $\emptyset$, $\emptyset$, $\emptyset$);

20: end if
Algorithm 4 TryMatchPDG(PDG1, PDG2, L_{NV}, L_{VM}, L_X)

1: Input: PDG1 and PDG2
2: Input: L_{NV}, list of (newConfigVar)
3: Input & Output: L_X, list of (varInG2 -> boolAssumption)
4: Input & Output: L_{VM}, list of (varInG1, varInG2)
5: Local: M_{PM} := ∅, map of
   (nodeInG1 -> lastFailedMatchInG2),
6: Local: L_{FA} := ∅, list of (nodeInG1 -> failedAssumInG2),
7: Local: S_{A} := ∅, stack of
   (nodeInG1 -> assumWhenMatchANodeInG1),
8: Output: S_{M}, Stack of (nodeInG1, nodeMatchInG2)
9: Push (Entry1, Entry2) to S_{M}
10: for each unmatched node n_{1} in preorder traversal of CFG
    of the function corresponding to PDG1 do
11:   n' := M_{PM}[n_{1}];
12:   for each unmatched node n_{2} that is (1) after n' in
      preorder traversal of CFG of the function
      corresponding to PDG2 and
      (2) (CtrlPred(n_{1}), CtrlPred(n_{2})) is in S_{M} do
13:     if isSameCanonicalOperator(n_{1}, n_{2})
14:        and noVarMatchingConflict(n_{1}, n_{2}, L_{VM})
15:        and isDataDepMatch(n_{1}, n_{2}) then
16:          push (n_{1}, n_{2}) in S_{M};
17:          updateVarMatch(n_{1}, n_{2}, L_{VM}); break;
18:       end if
19:     if isCanonicalOperatorWithAdditionalForms(n_{1}, n_{2})
20:        and (isNewFormalsWithMatchingConst(L_{X})
21:          or isNewFormalsWithMatchingVar(L_{X}))
22:        and noVarMatchingConflict(n_{1}, n_{2}, L_{VM})
23:        and isDataDepMatch(n_{1}, n_{2}) then
24:          push (n_{1}, n_{2}) in S_{M};
25:          updateVarMatch(n_{1}, n_{2}, L_{VM}); break;
26:       end if
27:     end for
28:     if isCondWithNewConfigVar(n_{2}, L_{NV}) then
29:       if existsAssum(n_{2} -> true, L_{X})
30:         restructPDG(n_{2} -> true, PDG2); continue;
31:       end if
32:       if noFailedAssum(n_{1} -> (n_{2} -> false), L_{FA}) then
33:         push n_{1} -> (n_{2} -> false) in S_{A};
34:         addVarAssum(n_{2} -> false, L_{X});
35:         restructPDG(n_{2} -> false, PDG2); continue;
36:       end if
37:     end if
38:   end for
39: end for
40: if isCondWithNewFormalParam(n_{2}) then
41:   if existsAssum(n_{2} -> true, L_{X}) then
42:     restructPDG(n_{2} -> true, PDG2); continue;
43:   end if
44:   if noFailedAssum(n_{1} -> (n_{2} -> false), L_{FA}) then
45:     push n_{1} -> (n_{2} -> false) in S_{A};
46:     addVarAssum(n_{2} -> false, L_{X});
47:     restructPDG(n_{2} -> false, PDG2); continue;
48:   end if
49:   if (false = existsAssum(n_{2} -> true, L_{X})) and
50:     noFailedAssum(n_{1} -> (n_{2} -> true), L_{FA}) then
51:     push n_{1} -> (n_{2} -> true) in S_{A};
52:     addVarAssum(n_{2} -> true, L_{X});
53:     restructPDG(n_{2} -> true, PDG2); continue;
54:   end if
55:   if (false = existsAssum(n_{2} -> false, L_{X})) and
56:     noFailedAssum(n_{1} -> (n_{2} -> false), L_{FA}) then
57:     push n_{1} -> (n_{2} -> false) in S_{A};
58:     addVarAssum(n_{2} -> false, L_{X});
59:     restructPDG(n_{2} -> false, PDG2); continue;
60:   end if
61: if notMatched(n_{1}) then
62:   if existsAssumForNode(S_{A}, n_{1}) then
63:     restructPDG(n_{2} -> true, PDG2); continue;
64:     end if
65:     (PDG2, S_A, L_{FA}) :=
66:     BacktrackAssum(n_{1}, PDG2, S_{A}, L_{FA});
67:     continue;
68:   end if
69: if stopMatch = Res then
70:   return (failure, ∅, ∅, ∅);
71: end if
Algorithm 5  BacktrackAssum($n_1$, PDG2, $S_A$, $L_{FA}$, $L_X$)

1: **Input**: $n_1$, a node in PDG1 without a match

2: **Input & Output**: PDG2

3: **Input & Output**: $S_A$, stack of (nodesInG1 → assumptionsInG2)

4: **Input & Output**: $L_{FA}$, list of (nodesInG1 → failedAssumptionsInG2)

5: **Input & Output**: $L_X$, list of (variableInG2 → boolAssumption)

6:

7: pop $n_1 \rightarrow (n_2 \rightarrow b)$ from $S_A$;

8: addFailAssum($n_1 \rightarrow (n_2 \rightarrow b)$, $L_{FA}$);

9: removeVarAssumIfLastReqNode(varInNode($n_2$), $L_X$);

10: for each $n_1 \rightarrow (n_3 \rightarrow b')$ in $L_{FA}$ where $n_3$ is after $n_2$ in preorder traversal of CFG of the function corresponding to PDG2 do

11: removeFailAssum($n_1 \rightarrow (n_3 \rightarrow b')$, $L_{FA}$);

12: end for

13: restorePDGForRemovedAssum(PDG2, $n_2 \rightarrow b$);

14: return (PDG2, $S_A$, $L_{FA}$, $L_X$);

Matching is the preorder traversal of the control flow graph (CFG) of the function for PDG1 and PDG2. This is because DFS-like traversal ensures more confidence in matching instead of BFS-like traversal. The matching case is shown from line 14 to 17. The matching includes the check of the canonical operator, variable matching conflict and dependence related to the node. The handling of the change “new configuration variables” [88] is shown from line 22 to 39. In brief, we match by trying either value of the new configuration variable or using the existing assumptions to configuration variables. We record failed assumptions to new configuration variable to avoid repetition assumptions. In a similar way, we handle the change “additional
Algorithm 6 BacktrackNodeMatch($S_M, M_{FM}, L_{VM}, L_{FA}, S_A$)

1: **Input & Output**: $S_M$, stack of (nodeInG1, nodeMatchInG2)

2: **Input & Output**: $M_{FM}$, map of (nodeInG1 $\mapsto$ lastFailedMatchInG2)

3: **Input & Output**: $L_{VM}$, list of (variableInG1, variableInG2)

4: **Input & Output**: $L_{FA}$, list of (nodesInG1 $\rightarrow$ failedAssumptionsInG2)

5: **Input & Output**: $S_A$, stack of (nodesInG1 $\rightarrow$ assumptionsInG2)

6: **Output**: Decision to continuing matching or not

7: 

8: if isEmpty($S_M$) or ((ENTRY1, ENTRY2) = $S_M$) then

9: return (stopMatch, PDG2, $\emptyset$, $\emptyset$, $L_{VM}$, $\emptyset$);

10: end if

11: ($n_1, n_2$) := pop $S_M$;

12: markAllAssumptionsForANodeAsFail($S_A, n_1, L_{FA}$);

13: $M_{FM}[n_1]$ := $n_2$;

14: removeVarMatchIfLastReqMatch($n_1, n_2, L_{VM}$);

15: for each $n_1' \rightarrow A$ in $L_{FA}$ where $n_1'$ is after $n_1$ in preorder traversal of CFG of the function corresponding to PDG1 do

16: clearAssumptions($n_1', L_{FA}$);

17: end for

18: for each $n_1' \mapsto n_2'$ in $M_{FM}$ where $n_1'$ is after $n_1$ in preorder traversal of CFG of the function corresponding to PDG1 do

19: clearLastFailedMatch($n_1', M_{FM}$);

20: end for

21: return (contMatch, $S_M, M_{FM}, L_{VM}, L_{FA}, S_A$);
formal parameters” 6.5 from line 40 to 58. The matching of function call sites with “additional formal parameters” is shown from line 18-21. If it is impossible to match a node in PDG1, then backtracking starts.

We then explain how the backtracking in algorithm 4 helps to avoid repeated matching and enumerate all possible matching. The matching includes enumerating all combinations of assumption and node matching, which is illustrated in Algorithm 5 and 6 respectively.

We explain how all combinations of true/false assumptions are enumerated with the presence of changes of new configuration variable or additional formal parameter. In lines 10 - 12 in Algorithm 5, whenever a failed assumption related to \( n_2 \) in PDG2 is recorded, we need to clear failed assumptions made in matching \( n_1 \) with nodes that are behind \( n_2 \) in preorder traversal of CFG of the function for PDG2. We use an example to illustrate how this helps enumerating all combinations of assumptions. Consider a change of “new configuration variable”, “\texttt{if} (x_1) \texttt{then} \{ \texttt{if} (x_2) \texttt{then} \{ \text{old,S} \} \}” where \( x_1 \) and \( x_2 \) are new configuration variables. When recording a failed assumption made towards “\texttt{if} (x_1)”, failed assumptions made to \texttt{if} (x_2) are removed. As a result, all four combination of assumptions could be enumerated.
In Algorithm 6, we show how to enumerate all possible node matching and avoid repeated node matching. We use a list $L_{FN}$ to record the last failed match for each node in PDG1. Whenever a backtrack starts, the last failed match for the PDG1 node $n_1$ (which is on the top of the stack $S_M$) is updated. In later matching for $n_1$, the matching candidates in PDG2 are only those that are behind the last failed match in the preorder traversal of the CFG of the function for PDG2. This is to ensure no repeated matching of one node in PDG2. On the other hand, upon backtracking, we clear the last failed match for any node $n'_1$ that is behind $n_1$ in the preorder traversal of the CFG for the function of PDG1. This is to allow matching for statement reordering. In this way, our matching algorithm ensures the trial for any possible combination of node matching and avoids repeated matching.

We next explain how to ensure consistent assumptions of new configuration variables. We use a list $L_X$ to record the current assumptions to new configuration variables. Our matching algorithm looks up the existing assumptions to new configuration variables before making a new assumption. If there exists an assumption for a configuration variable, then we check if the matching could proceed with that existing assumption.

The assumptions to additional formal parameters are handled in a similar way to the new configuration variables.

In algorithm 7, we show how to justify unmatched nodes in PDG2 after matching all nodes in PDG1. The justified updates include separation of states, exit-on-error, enumeration type extension and missing initialization [88]. Separation of state is identified by checking there is no data or control dependence from unmatched node to matched ones, which usually corresponds to computation for a removed log function call. Exit-on-error is identified by checking whether an unmatched if-statement is of form “If(C) then {...;exit();}” where “exit()” is an exit function. Similarly, the enu-
Algorithm 7 JustifyUnmatchedInPDG2(PDG2, L_{EN}, L_{EX}, L_{M})

1: **Input**: PDG2
2: **Input**: L_{EN}, list of (extended enumeration label)
3: **Input**: L_{EX}, list of (exitFunction)
4: **Input**: L_{M}, list of (nodeInPDG1, nodeMatchInPDG2)
5: **Local**: L_{J} := ∅, list of (justifiedUnmatchedNodeInG2)
6: **Output**: Decision to if all unmatched nodes in PDG2 are justified or not
7: 
8: **for each** unmatched node n_{2} in PDG2 in preorder traversal of CFG of the function corresponding to PDG2 do
9: 
10: \quad \text{Slice}_{n_{2}} := \text{createFwdPDGSlicingFrom}(n_{2}, \text{PDG2});
11: 
12: \quad \text{if} \ false = \text{hasMatchedNode(Slice}_{n_{2}}, L_{M}) \text{then}
13: \quad \quad \text{markJustifiedNodes(Slice}_{n_{2}}, L_{J}); \text{continue;}
14: \quad \quad \text{end if}
15: \quad \text{if} \ (\text{isOneBranchCond}(n_{2}) \text{ and hasExitAsCtrlDescendent}(n_{2}, L_{EX}))
16: \quad \quad \text{or} \ \text{isEnumExt}(n_{2}, \text{PDG2}, L_{EN})
17: \quad \quad \quad \text{or} \ \text{isMissingInit}(n_{2}, \text{PDG2}) \text{then}
18: \quad \quad \quad \text{markJustifiedNodeAndCtrlDescendents}(n_{2}, \text{PDG2}, L_{J}); \text{continue;}
19: \quad \quad \text{end if}
20: \text{return} \ \text{NotAllJustified;}
21: \text{end for}
22: \text{return} \ \text{AllJustified;}

133
meration type extension is identified by checking whether an If-statement is of form “If(x == L_{en}) then {...}” where $L_{en}$ is an extended label. The missing initialization is identified by checking whether the unmatched PDG node is for an assignment statement where the variable definition reaches and dominates at least one variable use.

6.4.5 Patch Generation

Patch generator produces the final patch based on the results of PDG matching. The patch generator produces state mapping as well as program point mapping.

1. The program point mapping is trivial if there is no statement reordering. With the presence of statement reordering, we forbid updates to occur within the block of statements between any two reordered statements. We identify the reordered statements by checking the order of mapped statements. Every statement is assigned with a unique number $N$ in pre-order traversal of the control flow graph of a function. If one statement $s_1$ maps to a statement $s'_1$, another statement $s_2$ maps to $s'_2$ where $(N(s_1) > N(s_2)) \land (N(s'_1) < N(s'_2))$, then $s_1$ and $s_2$ are reordered.

2. As to value copying in state mapping, we differentiate between simple typed variable, array variable and pointer variable. For a variable of unchanged struct type, we could do struct variable copy. However, for a reorganized structure type, we need to do value copy field by field. For array variable, we only support array extension and therefore we do the value copy using memcpy. To compile generated patch(C program), we have to include definitions of user-defined types in both the old program and the updated one.
In state mapping for changed functions, we limit the impact of unmatched functions in function call. If unmatched functions are encountered in the matching of their callers, we assume unmatched functions be matched and only note the unmatched functions.

6.5 Limitations

The current implementation has a number of obvious limitations. We list both non-fundamental limitations that can be addressed (with non-trivial effort) by extending the capabilities of the implementation and fundamental limitations that cannot be addressed.

Non-fundamental Limitations

- (Pointers) SitBack does not support mappings of recursive structures or heap data even for data that did not change. A more thorough treatment would require the use of shape analysis to generate state mappings for such structures.

- (Program Mapping) Our program mapping algorithm does not attempt to backtrack on assumptions across multiple functions. This proved to be adequate in the examples we tested, but a more general program mapping algorithm can be implemented.

- (Factoring) SitBack does not attempt to deal with functions that are split. Such an analysis can be done in principle using System dependence graphs.

- (CIL Preprocess) Due to the way CIL preprocesses code, there were cases when CIL introduced intermediate temporary in new but not the old version. Some of these had to be hand-edited to allow for a matching. Also, CIL moves some
static variables outside functions and renames them. These were also hand-edited.

- (Alias Analysis) Our alias analysis is not precise and a flow sensitive analysis [35] can result in smaller points-to sets.

**Fundamental Limitations**  The main fundamental limitation is tied to the impossibility of dealing with general functionality changes. Under this heading fall general bug fixes and algorithmic changes. This is not just a limitation of our approach but of any general attempt at calculating state mapping automatically. One way to deal with this limitation is to introduce finer-grain classification for the user which would enable the system to automate the mapping of more user-verified functions. Alternatively, one can envision making the user task simpler by providing an interactive tool to help in the generation of a mapping.

Despite the obvious limitations, it is important to note that **SitBack** is the only system for computing state mappings for non-trivial updates. Despite its limitations, it supports a large number of non-trivial updates and it proved capable of automatically handling changes that until now were not considered in the context of dynamic software updates. **SitBack** presents a significant advance in supporting dynamic software updates.

### 6.6 Evaluation

Our state mapping system is built on the CILv1.3.6 framework [76] and is entirely written in OCaml. All the experimentation is carried out in the Dell desktop OPTIPLEX 960 with Debian 4.1.1-21 (kernel 2.6.31) and gcc 4.1.2 20061115 prerelease. We use old Debian and gcc because UpStare is only known to be running on
this combination. Our evaluation is of two parts. We first evaluate the versatility of SitBack. We next evaluate the applicability of SitBack.

### 6.6.1 Versatility Evaluation

We evaluate the versatility of SitBack by testing it against consecutive updates of vsftpd. We chose vsftpd because of several reasons. First, vsftpd is widely used in practice and is widely studied in the DSU community [12, 74, 66, 28, 29, 75]. Second, SitBack is partially motivated by the observation of the evolution of vsftpd.

<table>
<thead>
<tr>
<th>Version</th>
<th>Total Changed</th>
<th>Generated</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.0 - 1.1.1</td>
<td>16</td>
<td>9</td>
<td>56.25%</td>
</tr>
<tr>
<td>1.1.1 - 1.1.2</td>
<td>8</td>
<td>2</td>
<td>25.00%</td>
</tr>
<tr>
<td>1.1.2 - 1.1.3</td>
<td>8</td>
<td>6</td>
<td>75.00%</td>
</tr>
<tr>
<td>1.1.3 - 1.2.0</td>
<td>64</td>
<td>18</td>
<td>28.13%</td>
</tr>
<tr>
<td>1.2.0 - 1.2.1</td>
<td>33</td>
<td>7</td>
<td>21.00%</td>
</tr>
<tr>
<td>1.2.1 - 1.2.2</td>
<td>10</td>
<td>4</td>
<td>40.00%</td>
</tr>
<tr>
<td>1.2.2 - 2.0.0</td>
<td>57</td>
<td>19</td>
<td>33.33%</td>
</tr>
<tr>
<td>2.0.0 - 2.0.1</td>
<td>7</td>
<td>5</td>
<td>71.43%</td>
</tr>
<tr>
<td>2.0.1 - 2.0.2</td>
<td>23</td>
<td>4</td>
<td>17.39%</td>
</tr>
<tr>
<td>2.0.2 - 2.0.3</td>
<td>18</td>
<td>8</td>
<td>44.44%</td>
</tr>
<tr>
<td>2.0.3 - 2.0.4</td>
<td>14</td>
<td>9</td>
<td>64.29%</td>
</tr>
<tr>
<td>2.0.4 - 2.0.5</td>
<td>21</td>
<td>11</td>
<td>52.38%</td>
</tr>
<tr>
<td>2.0.5 - 2.0.6</td>
<td>20</td>
<td>8</td>
<td>40.00%</td>
</tr>
<tr>
<td>total</td>
<td>299</td>
<td>110</td>
<td>36.79%</td>
</tr>
</tbody>
</table>

Table 6.1: Automatically Generated Mappings without Annotation
We have tested SitBack on 13 versions of vsftpd spanning 5.5 years worth of updates. The versions ranged from 8389 lines of code for the first version to more than 12,202 lines of code in the last version, a difference of 3813 lines of code. A total of 299 functions changed in the span of 13 versions, an average of about 12.75 lines of code per changed function. Another 638 functions did not changed textually but required a nontrivial stack transformer due to changes in structure data types that they use. For each version, we generated a mapping with the help of user annotation and user mappings (except for two cases that required no user mappings), applied the mappings and updated the running application using UpStare [67]. We collected statistics on the number of updated functions for each version change and the number of mappings that SitBack generates with and without annotations.

<table>
<thead>
<tr>
<th>Update Type</th>
<th>Number</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>log</td>
<td>17</td>
<td>12.41%</td>
</tr>
<tr>
<td>string constant</td>
<td>27</td>
<td>19.71%</td>
</tr>
<tr>
<td>type weakening</td>
<td>6</td>
<td>4.38%</td>
</tr>
<tr>
<td>generalization</td>
<td>8</td>
<td>5.84%</td>
</tr>
<tr>
<td>new condition</td>
<td>29</td>
<td>21.17%</td>
</tr>
<tr>
<td>separation of state</td>
<td>37</td>
<td>27.01%</td>
</tr>
<tr>
<td>bug fix</td>
<td>9</td>
<td>6.57%</td>
</tr>
<tr>
<td>renaming</td>
<td>4</td>
<td>2.92%</td>
</tr>
</tbody>
</table>

Table 6.2: Automatically Detected Update Types without Annotation

Without annotations (Table 6.1) an average of 36.79% of functions that are updated in all versions are automatically mapped. Some of these mappings still require user support to complete them. This is the case for separation of state in which the old PDG is completely matched but some new parts in the updated version are not
matched but do not interfere with the old part. For such cases, we consider that SitBack generates a mapping because it establishes the backward compatibility and matches all the variables from the old function. We notice that for specific updates, the percentage of functions that are automatically mapped tend to be lower for updates with many functions changed, but the lowest percentage is for an update with only 8 functions changed and 1 out of 8 is mapped. The categories of updates are shown in Table 6.2. We see that the updates are dominated with log functions, string constant changes, new conditions, and separation of state. Note that when counting categories of updates, we are double counting. So if a changed function has two categories of changes, they are counted separately in Table 6.2, but not in Table 6.1.

With annotation, the situation is considerably better. One update was fully mapped with the help of annotations. Annotations were used to ignore bug fixes, output formatting, complex refactoring and in some cases complex code rewriting involving new condition variables, but the updates were still backward compatible. We have stretched our annotation beyond what they were designed for by annotating both the old and the new version to ignore pieces of codes that are equivalent but different in the two versions. Out of 299 updated functions in 13 versions 58.86%
were automatically mapped with the help of annotations, a 60% improvement over the un-annotated case. Tables 6.3 and 6.4 shows the detailed statistics for the case of annotations.

<table>
<thead>
<tr>
<th>Version</th>
<th>Total Changed</th>
<th>Generated</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.0 - 1.1.1</td>
<td>16</td>
<td>11</td>
<td>68.75%</td>
</tr>
<tr>
<td>1.1.1 - 1.1.2</td>
<td>8</td>
<td>4</td>
<td>50.00%</td>
</tr>
<tr>
<td>1.1.2 - 1.1.3</td>
<td>8</td>
<td>7</td>
<td>87.25%</td>
</tr>
<tr>
<td>1.1.3 - 1.2.0</td>
<td>64</td>
<td>25</td>
<td>39.06%</td>
</tr>
<tr>
<td>1.2.0 - 1.2.1</td>
<td>33</td>
<td>22</td>
<td>67.00%</td>
</tr>
<tr>
<td>1.2.1 - 1.2.2</td>
<td>10</td>
<td>4</td>
<td>40.00%</td>
</tr>
<tr>
<td>1.2.2 - 2.0.0</td>
<td>57</td>
<td>32</td>
<td>56.14%</td>
</tr>
<tr>
<td>2.0.0 - 2.0.1</td>
<td>7</td>
<td>7</td>
<td>100.00%</td>
</tr>
<tr>
<td>2.0.1 - 2.0.2</td>
<td>23</td>
<td>9</td>
<td>39.13%</td>
</tr>
<tr>
<td>2.0.2 - 2.0.3</td>
<td>18</td>
<td>12</td>
<td>66.67%</td>
</tr>
<tr>
<td>2.0.3 - 2.0.4</td>
<td>14</td>
<td>11</td>
<td>78.57%</td>
</tr>
<tr>
<td>2.0.4 - 2.0.5</td>
<td>21</td>
<td>16</td>
<td>76.19%</td>
</tr>
<tr>
<td>2.0.5 - 2.0.6</td>
<td>20</td>
<td>16</td>
<td>80.00%</td>
</tr>
<tr>
<td>total</td>
<td>299</td>
<td>176</td>
<td>58.86%</td>
</tr>
</tbody>
</table>

Table 6.3: Automatically Generated Mappings with Annotation

Table 6.6 shows the categories of changes that were annotated (always with ignore comments); the percentages are rounded. The exception condition handling category refers to updates that introduce simple bug fixes that we do not automatically detect. By ignoring them, the user certifies that these are indeed safe to ignore at the time of the update. For example, a bug fix might check that the returned value of a function is positive. The new variable (struct field) init refers to initialization of new variable that
does not affect the matching (because it is guaranteed to be executed at the time of update for example). The unsupported condition category is for new conditions that involve testing string variables. The logging/output function represents a category in which the user certifies that the update results in changes to output format (including outputting new variables) that do not affect the semantics.

<table>
<thead>
<tr>
<th>Update Type</th>
<th>Number</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>log</td>
<td>17</td>
<td>10.24%</td>
</tr>
<tr>
<td>string constant</td>
<td>34</td>
<td>20.48%</td>
</tr>
<tr>
<td>type weakening</td>
<td>6</td>
<td>3.61%</td>
</tr>
<tr>
<td>generalization</td>
<td>8</td>
<td>4.82%</td>
</tr>
<tr>
<td>new condition</td>
<td>43</td>
<td>25.90%</td>
</tr>
<tr>
<td>separation of state</td>
<td>43</td>
<td>25.90%</td>
</tr>
<tr>
<td>bug fix</td>
<td>9</td>
<td>5.42%</td>
</tr>
<tr>
<td>renaming</td>
<td>6</td>
<td>3.61%</td>
</tr>
</tbody>
</table>

Table 6.4: Automatically Detected Update Types with Annotation

We measured the time to calculate the matching and, for most cases, it did not exceed 10 minutes to match all functions (with and without annotations) that could be matched in the program. Nonetheless, we notice that attempting to match some functions that could not be matched was taking an inordinate amount of time. This is due to the way we handle structures in functions. At the entry of a function, structures are flattened so that reaching definitions can be calculated by CIL and at the end the structure is reassembled before the return value. The structure flattening and reassembling is also carried out at every function call site. The structure flattening and reassembling only involve fields explicitly and directly referenced in a function. This creates a huge opportunity for reordering when a structure has some new fields,
which is not surprising given the brute-force nature of the backtracking algorithm. We set a threshold on the number of matching attempts before declaring failure. The matching algorithm keeps track for each node the number of times there is an attempt to match the node and terminates the matching if the threshold is exceeded.

The exact time of matching calculation for Vsftpd updates without annotation is shown in Table 6.5. “Size of Changes” refers to the size of changed functions in line of code, which reflects the total size of PDGs for matching. We used diff to facilitate the manual collection of changed functions in the two programs in one update and then used Cloc 1.64 [8] to count the lines of code of these changed functions in the two programs. Execution time is obtained by adding time calculation function around the PDG matching code.

Fig 6.7 shows the plot of PDG matching time as a function of the size of program changes. We noticed that the time of PDG matching is not proportional to the size of program changes. As is mentioned before, the cause of long matching time is due to a huge opportunity of statement reordering by structure flattening. We also noticed that, in cases of long matching time, most of the time is spent on matching “main” function in Vsftpd where a variable of the large vsf_session structure variable (with 34 fields in v1.1.0) is initialized. Because every field of the vsf_session structure is initialized in “main” function, the structure flattening and reassembling are required for all fields at every subfunction call site, which makes the PDG of “main” function with structure flattening really large. For example, the PDG (with structure flattening) of “main” function in Vsftpd v1.1.0 includes 430 nodes and 811 edges of all dependence types where 356 nodes and 670 edges are introduced by structure flattening and reassembling.

In matching “main” function in Vsftpd, the type and the initial value of a new field critically affect possibilities of reordering. If there are few existing fields with a
compatible type (in case of type weakening) and same initial value with respect to a new field, then there are limited possibilities of reordering.

<table>
<thead>
<tr>
<th>Update</th>
<th>Size of Changes (LOC)</th>
<th>PDG Matching Time (Secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.0 - 1.1.1</td>
<td>1227</td>
<td>334.96</td>
</tr>
<tr>
<td>1.1.1 - 1.1.2</td>
<td>646</td>
<td>7000.59</td>
</tr>
<tr>
<td>1.1.2 - 1.1.3</td>
<td>908</td>
<td>39.61</td>
</tr>
<tr>
<td>1.1.3 - 1.2.0</td>
<td>4922</td>
<td>10103.44</td>
</tr>
<tr>
<td>1.2.0 - 1.2.1</td>
<td>3728</td>
<td>439.59</td>
</tr>
<tr>
<td>1.2.1 - 1.2.2</td>
<td>923</td>
<td>54.47</td>
</tr>
<tr>
<td>1.2.2 - 2.0.0</td>
<td>3971</td>
<td>8631.58</td>
</tr>
<tr>
<td>2.0.0 - 2.0.1</td>
<td>94</td>
<td>86.82</td>
</tr>
<tr>
<td>2.0.1 - 2.0.2</td>
<td>2069</td>
<td>216.92</td>
</tr>
<tr>
<td>2.0.2 - 2.0.3</td>
<td>1488</td>
<td>117.80</td>
</tr>
<tr>
<td>2.0.3 - 2.0.4</td>
<td>1697</td>
<td>148.71</td>
</tr>
<tr>
<td>2.0.4 - 2.0.5</td>
<td>2181</td>
<td>293.88</td>
</tr>
<tr>
<td>2.0.5 - 2.0.6</td>
<td>3138</td>
<td>660.10</td>
</tr>
</tbody>
</table>

Table 6.5: PDG Matching Time for Vsftpd Updates without Annotation
If (e) then {
    If (e') then { p = funcA; }
    else { p = funcB; }
    p(1);
} else {
    If (e") then { p = funcC; }
    else { p = funcD; }
    p(0);
}

Figure 6.8: An Example of Function Pointer Usage

We collected the time of generating a patch and found most of the time is spent on alias analysis. Table 6.6 shows the time of our alias analysis for Vsftpd programs. We used Cloc 1.64 [8] to measure program size. The time of alias analysis is obtained by adding time counting functionality to our alias analysis code.

Fig 6.9 shows log-scale plot of alias analysis time as a function of program size. The time of alias analysis increases super-linearly with the size of the program. We have fixed point based solution of alias analysis and the convergence of our alias analysis depends on the program syntactic patterns. For example, consider a program in Fig 6.8. The function pointer p has a point-to set \{funcA, funcB, funcC, funcD\} by flow-insensitive analysis and the analysis of both call sites of p has to consider all elements in the point-to set of p. However, the alias analysis time is much less for a program that calls the four subfunctions directly. Other factors that may add complexity of our alias analysis include recursion, loop, heap object usage, size of program call graph.
Figure 6.9: Plot of Log2-Scale Alias Analysis Time for Vsftpd

<table>
<thead>
<tr>
<th>Version</th>
<th>Size of Program (LOC)</th>
<th>Alias Analysis Time (Secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.0</td>
<td>8389</td>
<td>16103.34</td>
</tr>
<tr>
<td>1.1.1</td>
<td>8468</td>
<td>16303.18</td>
</tr>
<tr>
<td>1.1.2</td>
<td>8731</td>
<td>17381.94</td>
</tr>
<tr>
<td>1.1.3</td>
<td>8839</td>
<td>18173.25</td>
</tr>
<tr>
<td>1.2.0</td>
<td>10011</td>
<td>24067.33</td>
</tr>
<tr>
<td>1.2.1</td>
<td>10506</td>
<td>26926.51</td>
</tr>
<tr>
<td>1.2.2</td>
<td>10547</td>
<td>25284.07</td>
</tr>
<tr>
<td>2.0.0</td>
<td>11527</td>
<td>26749.96</td>
</tr>
<tr>
<td>2.0.1</td>
<td>11543</td>
<td>26813.49</td>
</tr>
<tr>
<td>2.0.2</td>
<td>11612</td>
<td>27747.53</td>
</tr>
<tr>
<td>2.0.3</td>
<td>11743</td>
<td>28921.70</td>
</tr>
<tr>
<td>2.0.4</td>
<td>11857</td>
<td>30706.81</td>
</tr>
<tr>
<td>2.0.5</td>
<td>11923</td>
<td>33742.57</td>
</tr>
<tr>
<td>2.0.6</td>
<td>12202</td>
<td>37402.41</td>
</tr>
</tbody>
</table>

Table 6.6: Alias Analysis Time for Vsftpd
6.6.2 Applicability Evaluation

We show the effectiveness of automatic state mapping by testing DSU for five open source programs, sshd, vsftpd, cherokee, less, pexec. Sshd is the ssh daemon, vsftpd is ftp server, cherokee is http server, less is a linux command to show content of a file page by page, and pexec is a linux command that enables parallel execution of a program. All five programs are interactive, which means that a user could interact with these running programs.

For each of the five programs, we test one update. The testing process is as follows. We first merge the base and updated programs by UpStare [67] and CIL [76]. Then we feed the two merged programs to our SitBack tool to generate the auto state mapping. We next build the update patch using the auto state mapping and finally apply the update using UpStare. We ensure the success of update by checking the program version and have the basic use of the update programs. For example, we request web pages after updating cherokee, we download files after updating vsftpd; as to less/pexec, we check the program is still running after update because the update is minor.

The details of our experiments are shown in figure 6.10 and 6.11 below.

In Figure 6.10, “Tot.” refers to the total number of changed functions and “Map.” refers to changed functions that are auto mapped. SitBack tool fails to generate complete state mapping for all the changed functions in sshd / vsftpd / cherokee due to unsupported program update patterns. However the update experiment is still successful because the updated functions are not on the stack at the time of update. In our experiments, we do not intentionally restrict the update points in program. We do have restriction in the ways of applying the update. For example, the update of cherokee is applied at the time of no ongoing requests. Naturally we do not expect
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sshd 3.6.1p1 - p2</td>
<td>16</td>
<td>12</td>
<td>prompt msg improvement</td>
<td>If-predicate strengthen/weakening</td>
</tr>
<tr>
<td>vsftpd 2.0.0 - 2.0.1</td>
<td>7</td>
<td>5</td>
<td>var rename, prompt msg improvement, separation of state</td>
<td>stmt movement into If-stmt</td>
</tr>
<tr>
<td>cherokee 0.4.2-b1 - b2</td>
<td>12</td>
<td>8</td>
<td>separation of state (new var init/free)</td>
<td>function actual parameter extension</td>
</tr>
<tr>
<td>less 335 - 337</td>
<td>2</td>
<td>2</td>
<td>prompt msg improvement</td>
<td>N/A</td>
</tr>
<tr>
<td>pexec 1.0rc7 - rc8</td>
<td>3</td>
<td>3</td>
<td>array idx out of bound bugfix, prompt msg improvement</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Figure 6.10: Statistics of Automated State Mapping for Real World Software Update

an incomplete patch be successfully applied under all kinds of production workload. The highlight is that the automatic state mapping works in some cases of update.

Figure 6.11 shows the size and complexity of updated programs as well as the size of auto state mapping. In figure 6.11, “Bsize” refers to the size of base program in LOC; “# of bfunc.” refers to the number of functions in the base program which is a metric of program complexity; “Msize” refers to the size of automatic program state mapping in LOC. The program size or map size in Figure 6.11 is obtained by using CLoc 1.6.2 [8] against merged programs or automatic state mapping by CIL [76]. The “# of bfunc.” is also calculated against merged programs.
<table>
<thead>
<tr>
<th>Software Version</th>
<th>Bsize(LOC)</th>
<th># of bfunc.</th>
<th>Msize(LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sshd 3.6.1p1 - p2</td>
<td>37,604</td>
<td>808</td>
<td>558</td>
</tr>
<tr>
<td>vsftpd 2.0.0 - 2.0.1</td>
<td>23,838</td>
<td>476</td>
<td>223</td>
</tr>
<tr>
<td>cherokee 0.4.2-b1 - b2</td>
<td>12,890</td>
<td>209</td>
<td>233</td>
</tr>
<tr>
<td>less 335 - 337</td>
<td>31,245</td>
<td>390</td>
<td>62</td>
</tr>
<tr>
<td>pexec 1.0rc7 - rc8</td>
<td>26,831</td>
<td>202</td>
<td>162</td>
</tr>
</tbody>
</table>

Figure 6.11: Statistics of Base Programs and Auto-map for DSU

We show examples of mapped and missed program changes for each of the five programs below.

**Sshd**  Sshd is the daemon that allows users to do remote login and remote file transfer, which is widely used in practice [11]. Sshd is also widely studied by the DSU research community [74, 28, 29]. In our testing of successful sshd update, we do two checks. The first check is the version of running sshd program and the second check is to do ssh login after the update. Due to the incomplete state mapping, we are not able to apply the update and keep connected users function normally.

Examples of mapped and missed program changes are shown in figure 6.12 and 6.13.

In sshd update 3.6.1p1 - p2, the mapped program changes are all prompt message improvement, namely, string constant changes. These string constant changes are backward compatible obviously.

Two examples of missed program changes are shown in figure 6.13. In the changes in line 1 - 5 in figure 6.13, the If-predicate is strengthened by the use of the local variable pw. SitBack fails to identify the change as backward compatible because the execution of the two programs may differ based on the value of variable pw. Because
1: static void sshd_exchange_identification(int sock_in, int sock_out) { ...
2:     snprintf((buf), sizeof(buf), “SSH-%d.%d-%.100s\n”,
3: – major, minor, “OpenSSH_3.6.1p1”);
4: + major, minor, “OpenSSH_3.6.1p2”);
5:  ...
}

a: int main(int ac, char **av) { ...

b: – debug(“sshd version %.100s”, “OpenSSH_3.6.1p1”);

c: + debug(“sshd version %.100s”, “OpenSSH_3.6.1p2”); ...

d:  ...
}

Figure 6.12: Some Mapped Changes in Sshd 3.6.1p1 - p2

SitBack does not have the knowledge of variable pw at run time, the change is not
backward compatible by compile time check. A possible solution for such kind of
If-predicate changes is to identify the condition for the variable pw under which the
two If-predicates evaluate to “equivalent” value.

The changes in line a - i in figure 6.13 is to add more cases of return with the
detection of more exceptions and to direct all returns to one site by the switch variable
ok. Similar to the changes in line 1 -5 in figure 6.13, the program change can not be
made equivalent due to the different control flow. SitBack is detecting equivalence by
program syntactic check or manipulation. Therefore, SitBack fails to find the change
to be backward compatible. To find equivalence in different control flow, we need
to use other methodology used in [57, 39, 45]. The methodology in [57, 39, 45] is
Figure 6.13: Some Missed Changes in Sshd 3.6.1p1 - p2

to interpret programs into first order logic formula and then to use a logic solver to decide whether the two programs could be made equivalent.

Vsftpd  Vsftpd is an ftp server program widely used in a number of big websites like Ubuntu, CentOS, Fedora distribution sites [12]. In addition, vsftpd is widely studied in the DSU community [74, 66, 28]. In testing of vsftpd update, we ensure the successful update by checking the version number of the running ftp and continuing
operations of existing clients connected to the server. The surprise is that even
ongoing ftp operation like large file transfer is not affected by the update.

Examples of mapped and missed program changes are shown in figure 6.14 and 6.15.

In figure 6.14, we see that SitBack can handle variable renaming and additional
statement that has no data dependence to existing programs. It is straightforward to
see the backward compatibility with the presence of variable renaming and additional
statements.

Two similar examples of missed program changes in vsftpd update are shown in fig-
figure 6.15. In lines 2 - 6 in figure 6.15, existing statement “retval = str_netfd_read(p_dest,
fd, len)” is moved into an If-stmt where the If-predicate is checking the return value
of an prior function call. SitBack could not identify the change as backward compat-
1: void priv_sock_get_str(int fd, struct mystr* p_dest) { ...
2:   unsigned int len = (unsigned int) priv_sock_get_int(fd);
3:   - retval = str_netfd_read(p_dest, fd, len);
4:   + str_empty(p_dest);
5:   + if (len > 0) {
6:     + retval = str_netfd_read(p_dest, fd, len); ...
7:   }
8:   ...
9: }

a: void priv_sock_send_str(int fd, struct mystr const *p_str) { ...

b: - priv_sock_send_int(fd, (int) str_getlen(p_str));

c: + unsigned int len = str_getlen(p_str);

d: + priv_sock_send_int(fd, (int) len);

e: + if (len > 0) {

f:     str_netfd_write(p_str, fd);

g: + }  

h: ...  }

Figure 6.15: Some Missed Changes in Vsftpd 2.0.0 - 2.0.1
ible because the knowledge of the return value of the function call is not available at compile time. To handle such kind of changes, SitBack needs to relax the check of compatibility to allow the program matching. Also the SitBack needs to warn the user of the condition to apply the state mapping to ensure backward compatibility. The missed program change in lines a - h in figure 6.15 is similar to that in lines 1 - 7 in the same figure.

**Cherokee** Cherokee is a long running http server which performs better than Apache in cases [7]. When building cherokee program, we merged the cherokee library into the program to avoid the failure of update due to dependence inconsistency. The merge of cherokee library with cherokee program adds the complexity of state mapping because SitBack has to map the difference in the library. We ensure the success of update by two ways. We check the running cherokee server program by telnet and we also request existing html pages.

Some of the mapped and missed program changes by SitBack are shown in figure 6.16 and 6.17 respectively.

From figure 6.16, SitBack is able to detect functionality extension in the form of additional statements that have no data dependence to the existing programs. For example, the lines b - d in figure 6.16 shows the cleanup of new string variable “web_directory” that does not affect other existing statements in the function “cherokee_connection_free”.

SitBack fails to match the constant actual function parameter with non constant ones unless SitBack establishes the fact that the function calls with different actual parameters are backward compatible. Currently SitBack only supports the matching of additional formal parameters that are used as a switch to enable/disable of functionality extension. like the example shown in figure 5.4.
1: ret_t cherokee_connection_send_header (cherokee_connection_t *cnt) { ...
2: -  cherokee_buffer_add(cnt->buffer, “Server: Cherokee/0.4.21\r\n”, 24);
3: +  cherokee_buffer_add(cnt->buffer, “Server: Cherokee/0.4.22\r\n”, 24);
4: ... }

a: ret_t cherokee_connection_free (cherokee_connection_t *cnt) { ...

b: +  if (cnt->web_directory) {

c: +     free((void *)cnt->web_directory);

d: +     cnt->web_directory = 0; }

e: ... }

Figure 6.16: Some Mapped Changes in Cherokee 0.4.2b1 - b2

1: ret_t cherokee_connection_create_handler (cherokee_connection_t *cnt, ...) { ...
2: -  ret = cherokee_plugin_table_get(..., 0);
3: +  ret = cherokee_plugin_table_get(..., & cnt->web_directory);
4: ... }

Figure 6.17: Missed Change in Cherokee 0.4.2b1 - b2
To resolve the difference in figure 6.17, we need to evaluate the impact of different function calls. There are ways to do this. One solution is to do a slicing of the two called functions for the parameter and checks if the slicing match. The limitation of slicing matching is that the difference in executions with different parameters may not be shown in the difference of program slicing. A more sophisticated solution is to translate the called function into logic formula and employs a logic resolver to check if the difference function call are equivalent with reference to existing program equivalence detection tools [57, 39, 45].

**Conclusion** We showed the applicability of our state mapping tool SitBack using five real world programs. In each of the five program updates, SitBack generates complete or almost complete state mapping that is successfully applied dynamically. We have detailed discussion of mapped and missed program changes for each of the five program updates. We explained why mapped changes are backward compatible and why missed changes are not identified as backward compatible. We even suggest possible extension of SitBack for missed changes. Based on our evaluation, it is promising to use static inference tool to automatically generate state mapping that ensures backward compatibility. The limitation of static inference tool is that it fails to detect program equivalence in different control flows.
This chapter summarizes the dissertation and proposes the future work.

This dissertation aims to automate the solution of the state mapping problem in DSU. We include both a formal treatment of the state mapping problem and a practical study of the problem. We summarize the contributions of this dissertation below.

1. We propose a general formal definition of DSU safety: backward compatibility. We focus on the software update where the environment of a program execution may not be necessarily updated. To define backward compatibility formally, we systematically formalize program execution, execution environment, specification and dynamic software update.

2. We propose a framework of program equivalence for a formal programming language with explicit input and output statement. Our framework of program equivalence applies to non terminating execution as well as terminating execution. The framework allows several forms of update including statement reordering, loop fission or fusion, additional unrelated statements and potentially variable renaming.

3. We carried out a detailed study of real world program evolution for those that are widely used in practice, namely vsftpd, sshd, icecast. From the study, we summarize classes of update that are provably backward compatible.

4. We implemented a state mapping tool based on PDG matching, which assists developers to find a complete state mapping for a backward compatible DSU.
A detailed evaluation of our state mapping tool shows that it is promising to automate state mapping for patterns of program update.

7.1 Future Work

There are two main lines of inquiry for future extensions of the work. One part is for our formal treatment of the state mapping problem, the other is for the implementation of our state mapping tool.

There are several possible extensions to our formal study of the state mapping problem, including formalizing more update classes, extending our framework of equivalence for non-structure-preserving updates, or carrying out a study on more real-world program evolution.

In our study of real-world program updates, we notice that each update may contain various bug fixes. However, there is no particular pattern for general bug fixes. Though we could enumerate as many types of bug fixes as possible, this kind of treatment is tedious and does not help us understand the commonalities amongst various types of bug fixes. It is desirable to find a concise formal treatment for bug fixes.

Our current formal programming language includes several primitive types (e.g., int, long, enum) and language control structure (i.e., If-Branch, Loop). It is desirable to extend our language for simulating more practical program updates. For example, the pointer type and an arbitrary jump statement are common features in modern programming language. It is desirable to extend our formal language with pointer types and jump statements. The extension of our programming language will enable an expression of more practical update classes.

An extension of our programming language will probably lead to a redesign of our framework of equivalence. Furthermore, our framework of program equivalence
accommodates program updates that are mostly structure preserving. It is desirable to integrate proof rules for non structure preserving program updates such as proof rules in tools like [39, 57].

Currently, our treatment of state mapping is based on the study of evolution of several widely used programs. It is desirable to check how effective our methodology of state mapping is for more programs.

We find one possible extension for our state mapping tool. It is desirable to extend our state mapping tool for detecting non structure preserving equivalent program transformations. There are existing publications showing how to detect non structure preserving equivalent program transformation. For example, we could transform programs into logic formulas and employ an off-the-shelf logic solver to find conditions under which two programs are equivalent [57].
REFERENCES


[22] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the cight subset of the c language.


160


[46] Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. Specifying and verifying the correctness of dynamic software updates. VSTTE ’12, pages 278–293, 2012.


Figure A.1 shows an almost standard unsound and incomplete type system. The type system is unsound because of three reasons, (a) the possible value mismatch due to the subtype rule from the type Int to Long, (b) the implicit subtype between enumeration types and the type Long allowed by our semantics and (c) the possible array index out of bound. The type system is incomplete due to the parameterized “other” expressions. The notation Dom(Γ) borrowed from Cardelli [25] in rules Tvar1, Tvar2, Tlabels and Tfundecl refers to the domain of the typing environment Γ, which are identifiers bound to a type in Γ.
Figure A.1: Typing Rules
APPENDIX B

SUPPORTING LEMMAS FOR PROGRAM EQUIVALENCE

169
B.1 Properties of Imported Variables

**Lemma B.1.1.** \( \text{Imp}(S_1; S_2, X) = \text{Imp}(S_1, \text{Imp}(S_2, X)) \).

*Proof.* Let statement sequence \( S_2 = s_1; s_2; \ldots; s_k \) for some \( k > 0 \). The proof is by induction on \( k \). \( \square \)

**Corollary B.1.1.** \( \forall i \in \mathbb{Z}_+, \text{Imp}(S_i^{i+1}, X) = \text{Imp}(S, \text{Imp}(S^i, X)) \).

This is by lemma B.1.1.

**Lemma B.1.2.** \( \text{Imp}(S, A \cup B) = \text{Imp}(S, A) \cup \text{Imp}(S, B) \).

*Proof.* By structural induction on abstract syntax of statement sequence \( S \). \( \square \)

**Lemma B.1.3.** For statement \( s = \text{"while}(e)\{S\}" \) and a set of finite number of variables \( X \) such that \( X \cap \text{Def}(s) \neq \emptyset \), there is \( \beta > 0 \) such that \( \bigcup_{0 \leq i \leq (\beta+1)} \text{Imp}(S^i, X) \subseteq \bigcup_{1 \leq j \leq \beta} \text{Imp}(S^j, X) \).

*Proof.* By contradiction against the fact that is finite number of variables redefined in statement \( s \). \( \square \)

B.2 Properties of Expression Evaluation

We wrap the two properties of expression evaluation, which is based on the two properties of “other” expression evaluation. In the following, we use the notation \( \mathcal{E}' \) to expand the domain of the expression meaning function \( \mathcal{E}' : e \rightarrow \sigma \rightarrow (v_{\text{error}}, \{0, 1\}) \).

**Lemma B.2.1.** If every variable in \( \text{Use}(e) \) of an expression \( e \) has the same value w.r.t two value stores, the expression \( e \) evaluates to same value against the two value stores, \( (\forall x \in \text{Use}(e)) : \sigma_1(x) = \sigma_2(x) \Rightarrow (\mathcal{E}'[e]\sigma_1 = \mathcal{E}'[e]\sigma_2) \).

The proof is a case analysis of the expression \( e \).

**Lemma B.2.2.** If every variable in \( \text{Err}(e) \) of an expression \( e \) has same value w.r.t two pairs of (block, value store), \( \forall x \in \text{Err}(e) : \sigma_1(x) = \sigma_2(x) \) then one of the following holds:

1. the expression evaluates to crash against the two value stores, \( (\mathcal{E}'[e]\sigma_1 = (\text{error}, v_{\text{of}})) \land (\mathcal{E}'[e]\sigma_2 = (\text{error}, v_{\text{of}})) \);

2. the expression evaluates to no crash against the two pairs of (block, value store) \( (\mathcal{E}'[e]\sigma_1 \neq (\text{error}, v^1_{\text{of}})) \land (\mathcal{E}'[e]\sigma_2 \neq (\text{error}, v^2_{\text{of}})) \).

The proof is a case analysis of the expression \( e \).

With respect to Lemma B.2.1 and Lemma B.2.2, we extend semantic rule for expression evaluation as follows.
\[(r, m) \rightarrow (r', m')\]

\[\mathcal{E'} : e \rightarrow \sigma \rightarrow (v_{\text{error}} \times \{0, 1\})\]

\[\text{EEval'} \frac{j = 0}{(e, m(j, \sigma)) \rightarrow (\mathcal{E'}[e] \sigma, m)}\]

Figure B.1: Extended SOS Rule for Expressions

B.3 Properties of Remaining Execution

We assume that crash flag \(j = 0\) in given execution state \(m(j)\).

**Lemma B.3.1.** \((S_1, m) \rightarrow (S'_1, m') \Rightarrow (S_1; S_2, m) \rightarrow (S'_1; S_2, m')\).

The proof is by structural induction on abstract syntax of \(S_1\).

**Lemma B.3.2.** \((S_1, m) \xrightarrow{*} (S'_1, m') \Rightarrow (S_1; S_2, m) \xrightarrow{*} (S'_1; S_2, m')\).

By induction on number of steps \(k\) in execution \((S_1, m) \xrightarrow{k} (S'_1, m')\).

**Corollary B.3.1.** \((S_1, m) \xrightarrow{*} (\text{skip}, m') \Rightarrow (S_1; S_2, m) \xrightarrow{*} (S_2, m')\).

By lemma B.3.2 and rule Seq.

**Lemma B.3.3.** If one statement \(s\) is not in \(S\), then, after one step of execution \((S, m) \rightarrow (S', m')\), \(s\) is not in the \(S'\), \((s \notin S) \wedge ((S, m) \rightarrow (S', m')) \Rightarrow (s \notin S')\).

*Proof.* By induction on abstract syntax of \(S\). \(\square\)

**Lemma B.3.4.** If one statement \(s\) is not in \(S\), then, after the execution \((S, m) \xrightarrow{*} (S', m')\), \(s\) is not in the \(S'\), \((s \notin S) \wedge ((S, m) \xrightarrow{*} (S', m')) \Rightarrow (s \notin S')\).

*Proof.* By induction on the number \(k\) of the steps in the execution \((S, m) \xrightarrow{k} (S', m')\). \(\square\)

**Lemma B.3.5.** If a variable \(x\) is not defined in a statement sequence \(S\), then, after one step execution of \(S\), the value of \(x\) is not redefined, \((x \notin \text{Def}(S)) \wedge ((S, m(\sigma)) \rightarrow (S', m'(\sigma'))) \Rightarrow (x \notin \text{Def}(S')) \wedge (\sigma'(x) = \sigma(x))\).

By structural induction on abstract syntax of statement sequence \(S\), we show the lemma holds.

**Corollary B.3.2.** If a variable \(x\) is not defined in a statement sequence \(S\), then, after an execution of \(S\), the value of \(x\) is not redefined, \((x \notin \text{Def}(S)) \wedge (S, m(\sigma)) \rightarrow (S', m'(\sigma')) \Rightarrow (x \notin \text{Def}(S')) \wedge (\sigma'(x) = \sigma(x))\).

*Proof.* Let \((S, m) \xrightarrow{k} (S', m')\). The proof is by induction on \(k\) using lemma B.3.5. \(\square\)
Based on Corollary B.3.2, we extend the result to array variable elements.

**Corollary B.3.3.** If an element in an array variable \( x[i] \) is not defined in a statement sequence \( S \) in a program \( P = EN; V; S_{entry} \), then, after an execution of \( S \), the value of \( x[i] \) is not redefined, \( (x \notin \text{Def}(S)) \land ((x, i, \ast) \in \sigma) \land (S, m(\sigma)) \rightarrow (S', m'(\sigma')) \Rightarrow (x \notin \text{Def}(S')) \land \sigma'(x, i) = \sigma(x, i)) \).

**Lemma B.3.6.** If all of the following hold:
1. There is no loop of label \( n \) in statements \( S \), “while\( _n(e)\{S'\} \)” \( \notin S \);
2. The crash flag is not set, \( \dagger = 0 \);
3. There is an entry \( n \) in loop counter, \( (n, \ast) \in \text{loop}_c \);
4. There is one step execution, \( (S, m(\dagger, \text{loop}_c)) \rightarrow (S', m'(\text{loop}_c')) \);
then, \( \text{loop}_c'(n) = \text{loop}_c(n) \).

**Proof.** The proof is by induction on abstract syntax of \( S \), similar to that for lemma B.3.5. \( \square \)

**Corollary B.3.4.** If all of the following hold:
1. There is no loop of label \( n \) in statements \( S \), “while\( _n(e)\{S'\} \)” \( \notin S \);
2. The crash flag is not set, \( \dagger = 0 \);
3. There is an entry \( n \) in loop counter, \( (n, \ast) \in \text{loop}_c \);
4. There is multiple steps execution of stack depth \( d = 0 \), \( (S, m(\dagger, \text{loop}_c)) \rightarrow (S', m'(\text{loop}_c')) \);
then, \( \text{loop}_c'(n) = \text{loop}_c(n) \).

**Lemma B.3.7.** If all of the following hold:
1. A non-skip statement \( s \) is not in \( S \), \( (s \neq \text{skip}) \land (s \notin S) \);
2. There is one step execution of stack depth \( d = 0 \), \( (S, m) \rightarrow (S', m') \), then, \( s \notin S' \).

By structural induction on abstract syntax of statement sequence \( S \), we show the lemma holds.

**Lemma B.3.8.** Let \( s = \text{while}\_n(e)\{S''\} \). If both of the following hold:
- \( s \in S \);
- \( (S, m(\text{loop}_c)) \rightarrow (S', m'(\text{loop}_c')) \);

then one of the following holds:
1. The loop counter of label \( n \) is incremented by one, \( \text{loop}_c'(n) - \text{loop}_c(n) = 1 \);
2. There is no entry for label \( n \) in loop counter, \( (n, v) \notin \text{loop}_c \);
3. The loop counter of label \( n \) is not changed, \( \text{loop}_c'(n) - \text{loop}_c(n) = 0 \);

**Proof.** Let \( S = s'; S'' \). The proof is by induction on abstract syntax of \( s' \). \( \square \)
APPENDIX C

THE PROOF FOR EQUIVALENT TERMINATING COMPUTATION THEOREM
Theorem 1: If \( s_1 \) and \( s_2 \) are simple statements that satisfy the proof rule for equivalent computation of \( x \), \( s_1 \equiv s_2 \), and their initial states \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) agree on the values of the imported variables relative to \( x \), then \( s_1 \) and \( s_2 \) equivalently compute \( x \) when started in states \( m_1 \) and \( m_2 \) respectively, \((s_1, m_1) \equiv_x (s_2, m_2)\).

Proof.

The proof is a case analysis according to the cases in the definition of the proof rule for equivalent computation (i.e., Definition 11).

1. \( s_1 = s_2 \)

Since the two statements are identical, they have the same imported variables. By assumption, the imported variables of \( s_1 \) and \( s_2 \) have the same initial values, so it is enough to show that the value of \( x \) at the end of the computation only depends on the initial values of the imported variables.

(a) \( s_1 = s_2 = \text{“skip”} \). In this case, the states before and after the execution of skip are the same and \( \text{Imp}(\text{skip}, \{x\}) = \{x\} \).

(b) \( s_1 = s_2 = \text{lval} := e \).

i. \( \text{lval} = x \).

\( s_1 = s_2 = x := e \). By the definition of imported variables, \( \text{Imp}(x := e, \{x\}) = \text{Use}(e) \). The execution of \( s_1 \) proceeds as follows.

\[
\begin{align*}
(x := e, m(\sigma)) \\
&\to (x := E[e][\sigma, m(\sigma)]) \text{ by the EEval’ rule} \\
&\to (\text{skip}, m(\text{E}[e][\sigma/x])) \text{ by the Assign rule.}
\end{align*}
\]

The value of \( x \) after the full execution is \( \sigma[(\text{E}[e][\sigma])/x](x) \) which only depend on the initial values of the imported variables by the property of the expression meaning function.

ii. \( \text{lval} \neq \text{id} \).

By the definition of imported variables, \( \text{Imp}(s_1, \{x\}) = \text{Imp}(s_2, \{x\}) = \{x\} \). It follows, by assumption, that \( \sigma_1(x) = \sigma_2(x) \) and also \( s_1 \) terminate, \((s_1, m_1(\sigma_1)) \to (\text{skip}, m'_1(\sigma'_1)) \). Hence, \( \sigma'_1(x) = \sigma_1(x) \) by Corollary B.3.2. Similarly, \( s_2 \) terminates, \((s_2, m_2(\sigma_2)) \to (\text{skip}, m'_2(\sigma'_2)) \) and \( \sigma'_2(x) = \sigma_2(x) \). Therefore, \( \sigma'_2(x) = \sigma_2(x) = \sigma_1(x) = \sigma'_1(x) \) and the theorem holds.

(c) \( s_1 = s_2 = \text{“input \ id”} \).

i. \( x \in \text{Def(input \ id)} = \{\text{id}, \text{id}_I, \text{id}_{IO}\} \).

By the In rule, the execution of input \( \text{id} \) is the following.

\[
\begin{align*}
(\text{input \ id}, m(\sigma)) \\
&\to (\text{skip}, m(\text{tl}(\sigma(\text{id}_I))/\text{id}_I) \\
&\quad [\text{“} \sigma(\text{id}_{IO}) \cdot \text{hd}(\sigma(\text{id}_I))/\text{id}_{IO} \text{“}]) \text{).}
\end{align*}
\]

The value of \( x \) after the execution of “input \( \text{id} \)” is one of the following:

A. \( \text{tl}(\sigma(\text{id}_I)) \) if \( x = \text{id}_I \).
B. \( \sigma_1(\text{id}_{IO}) \cdot \text{hd}(\sigma(\text{id}_I)) \) if \( x = \text{id}_{IO} \).
C. \( \text{hd}(\sigma(\text{id}_I)) \) if \( x = \text{id} \).
By the definition of imported variables, \( \text{Imp}(\text{id}, \{x\}) = \{\text{id}_{IO}, \text{id}_I\} \).
So, in all cases, the value of \( x \) only depends on the initial values of the
imported variables \( \text{id}_I \) and \( \text{id}_{IO} \).

ii. \( x \notin \text{Def}(\text{id}) = \{\text{id}, \text{id}_I, \text{id}_{IO}\} \).

By same argument in the subcase \( \text{id} \neq x \) of case \( s_1 = s_2 = \text{"id} := e \),
the theorem holds.

(d) \( s_1 = s_2 = \text{"output} e \).

i. \( x = \text{id}_{IO} \)

By the definition of imported variables, \( \text{Imp}(\text{output} e, \{x\}) = \{\text{id}_{IO}\} \cup \text{Use}(e) \).
The execution of \( s_1 \) proceeds as follows.

\[
\begin{align*}
(\text{output} e, m(\sigma)) & \rightarrow (\text{output} E[e]_\sigma, m(\sigma)) \\
& \rightarrow (\text{skip}, m(\sigma[\text{"sigma} id_{IO} \cdot E[e]_\sigma / id_{IO}])).
\end{align*}
\]

The value of \( x \) after the execution is \( \sigma(id_{IO}) \cdot E[e]_\sigma \), which only
depends on the initial value of the imported variables of the statement
"output e" by the expression meaning function.

ii. \( x \neq \text{id}_{IO} \)

By same argument in the subcase \( \text{id} \neq x \) of case \( s_1 = s_2 = \text{"id} := e \),
the theorem holds.

2. \( s_1 \neq s_2 \)

(a) \( s_1 = \text{"input} id_1 \), \( s_2 = \text{"input} id_2 \), \( x \notin \{id_1, id_2\} \).

i. \( x \in \{id_1, id_{IO}\} \).

By the definition of imported variables, \( \text{Imp}(s_1, \{x\}) = \text{Imp}(s_2, \{x\}) = \{id_{IO}, id_I\} \).
It follows, by assumption, that \( \sigma(y) = \sigma(y), \forall y \in \{id_{IO}, id_I\} \).
The execution of \( s_1 \) proceeds as follows.

\[
\begin{align*}
(s_1, m_1) & = (\text{input} id_1, m_1(\sigma)) \\
& \rightarrow (\text{skip}, m_1(\sigma_1[tl(\sigma_1(id_I))/id_I]) \\
& [\text{"sigma} id_{IO} \cdot \text{hd}(\sigma_1(id_I))/id_{IO}] [\text{hd}(\sigma_1(id_I))/id_I])
\end{align*}
\]

Let \( \sigma'_1 = \sigma_1[tl(\bar{v})/id_I, \text{"sigma} id_{IO} \cdot \text{hd}(\bar{v})/id_{IO}, \text{hd}(\bar{v})/id_I] \). The value of
\( x \) after the execution of \( s_1 \) is one of the following:

A. \( \sigma'_1(x) = \text{tl}(\sigma_1(id_I)) \) if \( x = id_I \),
B. \( \sigma'_1(x) = \sigma_1(id_{IO}) \cdot \text{hd}(\sigma_1(id_I)) \) if \( x = id_{IO} \).

Similarly, \( (s_2, m_2) \rightarrow (\text{skip}, m_2(\sigma_2[tl(\sigma_2(id_I))/id_2]) \\
[\text{"sigma} id_{IO} \cdot \text{hd}(\sigma_2(id_2))/id_{IO}] [\text{hd}(\sigma_2(id_2))/id_2]).
\]

Let \( \sigma'_2 = \sigma_2[tl(\sigma_2(id_I))/id_I] [\text{"sigma} id_{IO} \cdot \text{hd}(\sigma_2(id_I))/id_{IO}] [\text{hd}(\sigma_2(id_I))/id_2] \).
Then the value of \( x \) after the execution of \( s_2 \) is one of the following:

A. \( \sigma'_2(x) = \text{tl}(\sigma_2(id_I)) \) if \( x = id_I \)
B. \( \sigma'_2(x) = \sigma_2(id_{IO}) \cdot \text{hd}(\sigma_2(id_I)) \) if \( x = id_{IO} \)

Repeatedly, \( \sigma_2(id_I) = \sigma_1(id_I) \) and \( \sigma_2(id_{IO}) = \sigma_1(id_{IO}) \).
Therefore, the theorem holds.
ii. $x \notin \{id_1, id_2\}$

Repeatedly, $x \notin \{id_1, id_2\}$. By same argument in the subcase $id \neq x$ of case $s_1 = s_2 = "id := e"$, the theorem holds.

(b) all the above cases do not hold and $x \notin \text{Def}(s_1) \cup \text{Def}(s_2)$

By same argument in the subcase $id \neq x$ of case $s_1 = s_2 = "id := e"$, the theorem holds.

Theorem 2: If statement sequence $S_1$ and $S_2$ satisfy the proof rule of equivalent computation of a variable $x$, $S_1 \equiv_x S_2$, and their initial states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ agree on the initial values of the imported variables relative to $x$, $\forall y \in \text{Imp}(S_1, \{x\}) \cup \text{Imp}(S_2, \{x\}) : \sigma_1(y) = \sigma_2(y)$, then $S_1$ and $S_2$ equivalently compute the variable $x$ when started in state $m_1$ and $m_2$ respectively, $(S_1, m_1) \equiv_x (S_2, m_2)$.

Proof.

By induction on $\text{size}(S_1) + \text{size}(S_2)$, the sum of the program size of $S_1$ and $S_2$.

Base case.

$S_1 \equiv_x S_2$ where $S_1$ and $S_2$ are two simple statements. This theorem holds by theorem 1.

Induction step

The hypothesis IH is that Theorem 2 holds when $\text{size}(S_1) + \text{size}(S_2) = k \geq 2$.

Then we show that the Theorem holds when $\text{size}(S_1) + \text{size}(S_2) = k + 1$. The proof is a case analysis according to the cases in the definition of the proof rule of terminating computation of statement sequence. the two big categories enum

1. $S_1$ and $S_2$ are one statement such that one of the following holds:

   (a) $S_1$ and $S_2$ are If statement that define the variable $x$:

   $S_1 = \text{"If (e) then } \{S'_1\} \text{ else } \{S''_1\} \text{"}$, $S_2 = \text{"If (e) then } \{S'_2\} \text{ else } \{S''_2\} \text{"}$

   such that all of the following hold:

   - $x \in \text{Def}(S_1) \cap \text{Def}(S_2)$;
   - $S'_1 \equiv_x S'_2$;
   - $S''_1 \equiv_x S''_2$;

   We first show that the evaluations of the predicate expression of $S_1$ and $S_2$ produce the same value when started from state $m_1(\sigma_1)$ and $m_2(\sigma_2)$, w.l.o.g. say zero. Next, we show that $S'_1$ started in the state $m_1$ and $S'_2$ in the state $m_2$ equivalently compute the variable $x$.

   In order to show that the evaluations of predicate expression of $S_1$ and $S_2$ produce same value when started from state $m_1(\sigma_1)$ and $m_2(\sigma_2)$, we show that the variables used in predicate expression of $S_1$ and $S_2$ are a subset of imported variables in $S_1$ and $S_2$ relative to $x$. This is true by the definition of imported variables, $\text{Use}(e) \subseteq \text{Imp}(S_1, \{x\})$, $\text{Use}(e) \subseteq \text{Imp}(S_2, \{x\})$. By assumption, the value stores $\sigma_1$ and $\sigma_2$ agree on the values of the variables used in predicate expression of $S_1$ and $S_2$, $\sigma_1(y) = \sigma_2(y), \forall y \in \text{Use}(e)$. By the property of expression meaning function $E$, the predicate expression of $S_1$ and $S_2$ evaluate to the same value when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$, $E[e] \sigma_1 = E[e] \sigma_2$, w.l.o.g $E[e] \sigma_1 = E[e] \sigma_2 = (0, v_{ef})$. Then the execution of $S_1$ proceeds as follows.
\[(S_1, m_1(\sigma_1))\]
\[= (\text{If } (e) \text{ then } \{S'_1\} \text{ else } \{S''_1\}, m_1(\sigma_1))\]
\[\to (\text{If } ((0, \tau_{of})) \text{ then } \{S'_1\} \text{ else } \{S''_1\}, m_1(\sigma_1))\]
\[\text{by the EEval' rule}\]
\[\to (\text{If } (0) \text{ then } \{S'_1\} \text{ else } \{S''_1\}, m_1(\sigma_1))\]
\[\text{by the E-Oflow1 or E-Oflow2 rule}\]
\[\to (S'_1, m_1(\sigma_1))\]

Similarly, the execution from \((s_2, m_2(\sigma_2))\) gets to \((S''_2, m_2(\sigma_2))\).

By the hypothesis IH, we show that \(S'_1\) and \(S''_1\) compute the variable \(x\) equivalently when started in state \(m_1(\sigma_1)\) and \(m_2(\sigma_2)\) respectively. To do that, we show that all required conditions are satisfied for the application of hypothesis IH.

- \(\text{size}(S'_1) + \text{size}(S''_1) < k\).
  Because \(\text{size}(S_1) = 1 + \text{size}(S'_1) + \text{size}(S''_1)\), \(\text{size}(S_2) = 1 + \text{size}(S'_2) + \text{size}(S''_2)\).

- the value stores \(\sigma_1\) and \(\sigma_2\) agree on the values of the imported variables in \(S'_1\) and \(S'_2\) relative to \(x\), \(\sigma_1(y) = \sigma_2(y), \forall y \in \text{Imp}(S'_1, \{x\}) \cup \text{Imp}(S'_2, \{x\})\).
  By the definition of imported variables,
  \[\text{Imp}(S'_1, \{x\}) \subseteq \text{Imp}(S_1, \{x\}), \text{Imp}(S'_2, \{x\}) \subseteq \text{Imp}(S_2, \{x\})\).

By the hypothesis IH, \(S'_1\) and \(S''_1\) compute the variable \(x\) equivalently when started in state \(m_1(\sigma_1)\) and \(m_2(\sigma_2)\) respectively. Therefore, the theorem holds.

(b) \(S_1\) and \(S_2\) are while statement that define the variable \(x\):
\(S_1 = \text{"while}_{(n_1)}(e) \ {S''_1}\), \(S_2 = \text{"while}_{(n_2)}(e) \ {S''_2}\)" such that both of the following hold:

- \(x \in \text{Def}(S_1) \cap \text{Def}(S_2)\);
- \(S''_1 \equiv^S S''_2\) for \(\forall y \in \text{Imp}(S_1, \{x\}) \cup \text{Imp}(S_2, \{x\})\).

By Lemma 4.3.2, we show \(S_1\) and \(S_2\) compute the variable \(x\) equivalently when started from state \(m_1(m^1, \sigma_1)\) and \(m_2(m^2, \sigma_2)\) respectively. The point is to show that all required conditions are satisfied for the application of lemma 4.3.2.

- loop counter value of \(S_1\) and \(S_2\) are zero.
  By our assumption, the loop counter value of \(S_1\) and \(S_2\) are initially zero.
- \(S_1\) and \(S_2\) have same imported variables relative to \(x\), \(\text{Imp}(S_1, \{x\}) = \text{Imp}(S_2, \{x\}) = \text{Imp}(\Delta)\).
  This is obtained by Lemma 4.3.3.
- the initial value store \(\sigma_1\) and \(\sigma_2\) agree on the values of the imported variables in \(S_1\) and \(S_2\) relative to \(x\), \(\sigma_1(y) = \sigma_2(y), \forall y \in \text{Imp}(S_1, \{x\}) \cup \text{Imp}(S_2, \{x\})\).
  By assumption, this holds.

177
• $S_1''$ and $S_2''$ compute the imported variables in $S_1$ and $S_2$ relative to $x$ equivalently, $(S_1'', m_{S_1''}(\sigma_{S_1''})) \equiv_y (S_2'', m_{S_2''}(\sigma_{S_2''}))$, $\forall y \in \text{Imp}(\Delta)$ with value stores $\sigma_{S_1''}$ and $\sigma_{S_2''}$ agreeing on the values of the imported variables in $S_1''$ and $S_2''$ relative to $\text{Imp}(\Delta)$, $\sigma_{S_1''}(z) = \sigma_{S_2''}(z), \forall z \in \text{Imp}(S_1'', \text{Imp}(\Delta)) \cup \text{Imp}(S_2'', \text{Imp}(\Delta))$.

By the definition of program size, the sum of the program size of $S_1'$ and $S_2'$ is less than $k$, size($S_1'$) + size($S_2'$) < $k$. By the hypothesis IH, $S_1''$ and $S_2''$ compute the imported variables in $S_1$ and $S_2$ relative to $x$ equivalently when started in states $m_{S_1''}(\sigma_{S_1''})$ and $m_{S_2''}(\sigma_{S_2''})$ with value store $\sigma_{S_1''}$ and $\sigma_{S_2''}$ agreeing on the values of the imported variables in $S_1''$ and $S_2''$ relative to the variables $\text{Imp}(\Delta)$.

By Lemma 4.3.2, we show $S_1$ and $S_2$ compute the variable $x$ equivalently when started from state $m_1(m_1^1, \sigma_1)$ and $m_2(m_2^2, \sigma_2)$ respectively. The theorem holds.

(c) $S_1$ and $S_2$ do not define the variable $x$: $x \notin \text{Def}(S_1) \cup \text{Def}(S_2)$.

By the definition of imported variable, the imported variables in $S_1$ and $S_2$ relative to $x$ are both $x$, $\text{Imp}(S_1, \{x\}) = \text{Imp}(S_2, \{x\}) = \{x\}$. By assumption, the initial values $\sigma_1$ and $\sigma_2$ agree on the value of the variable $x$, $\sigma_1(x) = \sigma_2(x)$. In addition, by assumption, execution of $S_1$ and $S_2$ when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ terminate, $(S_1, m_1(\sigma_1)) \overset{*}{\rightarrow} (\text{skip}, m_1'(\sigma_1'))$, $(S_2, m_2(\sigma_2)) \overset{*}{\rightarrow} (\text{skip}, m_2'(\sigma_2'))$. Finally, by Corollary B.3.2, the value of $x$ is not changed in execution of $S_1$ and $S_2$, $\sigma_1'(x) = \sigma_1(x) = \sigma_2(x) = \sigma_2'(x)$. The theorem holds.

2. $S_1$ and $S_2$ are not both one statement such that one of the following holds:

(a) Last statements both define the variable $x$ such that all of the following hold:

- $S_1' \equiv^S_y S_2', \forall y \in \text{Imp}(s_1, \{x\}) \cup \text{Imp}(s_2, \{x\})$;
- $x \in \text{Def}(s_1) \cap \text{Def}(s_2)$;
- $s_1 \equiv^S_x s_2$.

We show that $S_1'$ and $S_2'$ compute the imported variables in $s_1$ and $s_2$ relative to the variable $x$ equivalently when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively by the hypothesis IH. To do that, we show the required conditions are satisfied for applying the hypothesis IH.

- size($S_1'$) + size($S_2'$) < $k$.
  By the definition of program size, size($s_1$) ≥ 1, size($s_2$) ≥ 1. Hence, size($S_1'$) + size($S_2'$) < $k$.
- the executions from $(S_1, m_1(\sigma_1))$ and $(S_2, m_2(\sigma_2))$ terminate respectively,
  $(S_1', m_1(\sigma_1)) \overset{*}{\rightarrow} (\text{skip}, m_1''(\sigma_1''))$, $(S_2', m_2(\sigma_2)) \overset{*}{\rightarrow} (\text{skip}, m_2''(\sigma_2''))$.

By assumption, the execution from $(S_1, m_1(\sigma_1))$ and $(S_2, m_2(\sigma_2))$ terminate, then the execution of $S_1'$ and $S_2'$ from state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ terminate, $(S_1', m_1(\sigma_1)) \overset{*}{\rightarrow} (\text{skip}, m_1''(\sigma_1''))$, $(S_2', m_2(\sigma_2)) \overset{*}{\rightarrow} (\text{skip}, m_2''(\sigma_2''))$.
• the initial value stores agree on the values of the variables:
\[ \text{Imp}(S'_1, \text{Imp}(s_1, \{ x \})) \cup \text{Imp}(S'_2, \text{Imp}(s_2, \{ x \})). \]

By Lemma 4.3.3, \( s_1 \) and \( s_2 \) have the same imported variables relative to \( x \), \( \text{Imp}(s_1, \{ x \}) = \text{Imp}(s_2, \{ x \}) = \text{Imp}(x) \). By the definition of imported variables, imported variables in \( S'_1 \) relative to \( \text{Imp}(x) \) are same as the imported variables in \( S_1 \) relative to \( x \), \( \text{Imp}(S'_1, \text{Imp}(s_1, \{ x \})) = \text{Imp}(S_1, \{ x \}) \). Similarly, \( \text{Imp}(S'_2, \text{Imp}(s_2, \{ x \})) = \text{Imp}(S_2, \{ x \}) \). Then, by assumption, the initial value stores agree on the values of the variables \( \text{Imp}(S'_1, \text{Imp}(s_1, \{ x \})) \) and \( \forall y \in \text{Imp}(S'_1, \text{Imp}(s_1, \{ x \})) \cup \text{Imp}(S'_2, \text{Imp}(s_2, \{ x \})), \sigma_1(y) = \sigma_2(y) \).

By the hypothesis IH, after the full execution of \( S'_1 \) from state \( m_1(\sigma_1) \) and the execution of \( S'_2 \) from state \( m_2(\sigma_2) \), the value stores agree on the values of the imported variables in \( s_1 \) and \( s_2 \) relative to \( x \), \( \sigma_1''(y) = \sigma_2''(y), \forall y \in \text{Imp}(x) = \text{Imp}(s_1, \{ x \}) = \text{Imp}(s_2, \{ x \}) \).

Then, we show \( s_1 \) and \( s_2 \) compute \( x \) equivalently. By Corollary B.3.1, \( s_1 \) and \( s_2 \) continue execution after the full execution of \( S'_1 \) and \( S'_2 \) respectively, \( (S'_1; s_1, m_1(\sigma_1)) \xrightarrow{\ast} (s_1, m''_1(\sigma''_1)) \), \( (S'_2; s_2, m_2(\sigma_2)) \xrightarrow{\ast} (s_2, m''_2(\sigma''_2)) \). When \( s_1 \) and \( s_2 \) are while statements, by our assumption of unique loop labels, \( s_1 \) is not in \( S'_1 \). By Corollary B.3.4, the loop counter value of \( s_1 \) is not redefined in the execution of \( S'_1 \). Similarly, the loop counter value of \( s_2 \) is not redefined in the execution of \( S'_2 \). By the hypothesis IH again, after the full execution of \( s_1 \) and \( s_2 \), the value stores agree on the value of \( x \), \( (s_1, m''_1(\sigma''_1)) \xrightarrow{\ast} (\text{skip}, m'_1(\sigma'_1)), (s_2, m''_2(\sigma''_2)) \xrightarrow{\ast} (\text{skip}, m'_2(\sigma'_2)) \) such that \( \sigma'_1(x) = \sigma'_2(x) \). The theorem holds.

(b) One last statement does not define the variable \( x \): W.l.o.g., \( (x \notin \text{Def}(s_2)) \land (S_1 \equiv S'_2) \).

We show that \( S_1 \) and \( S'_2 \) compute the variable \( x \) equivalently when started from state \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) by the hypothesis IH. First, by the definition of program size, \( \text{size}(s_2) \geq 1 \). Hence, \( \text{size}(S_1) + \text{size}(S'_2) \leq k \).

Next, by the definition of imported variables, \( \text{Imp}(S'_2, \{ x \}) \subseteq \text{Imp}(S_2, \{ x \}) \).

By assumption, \( \sigma_1(y) = \sigma_2(y) \) for \( \forall y \in \text{Imp}(S'_2, \{ x \}) \cup \text{Imp}(S_1, \{ x \}) \).

By the hypothesis IH, \( S_1 \) and \( S'_2 \) compute the variable \( x \) equivalently when started in state \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) respectively, \( (S'_2; s_2, m_2(\sigma_2)) \xrightarrow{\ast} (\text{skip}, m'_2(\sigma'_2)) \), \( (S_1, m_1(\sigma_1)) \xrightarrow{\ast} (\text{skip}, m'_1(\sigma'_1)) \) such that \( \sigma'_1(x) = \sigma'_2(x) \).

Then, we show that \( S_1 \) and \( S'_2 \) compute the variable \( x \) equivalently after the full execution of \( s_2 \). By Corollary B.3.1, \( s_2 \) continues execution immediately after the full execution of \( S'_2 \), \( (S'_2; s_2, m_2) \xrightarrow{\ast} (s_2, m''_2(\sigma''_2)) \). By assumption, the execution from \( (S'_2; s_2, m_2) \) terminates, \( (s_2, m''_2(\sigma''_2)) \xrightarrow{\ast} (\text{skip}, m'_2(\sigma'_2)) \).

By Corollary B.3.2, the value of \( x \) is not changed in the execution of \( s_2 \), \( \sigma'_2(x) = \sigma''_2(x) \). Hence, \( \sigma'_1(x) = \sigma'_2(x) \). The theorem holds.

(c) There are statements moving in/out of If statement:

\( s_1 = \text{If} \ (e) \text{ then } \{ S'_1 \} \text{ else } \{ S'_1 \}^*, \ s_2 = \text{If} \ (e) \text{ then } \{ S'_2 \} \text{ else } \{ S'_2 \}^* \)

such that none of the above cases hold and all of the following hold:

- \( S'_1 \equiv S'_2 \) for \( \forall y \in \text{Use}(e) \);
• $S'_1; S'_1 \equiv^S_x S'_2; S'_2$;
• $S'_1; S'_1 \equiv^S_x S'_2; S'_2$;
• $x \in \text{Def}(s_1) \cap \text{Def}(s_2)$;

Repeatedly $S_1 = S'_1; s_1, S_2 = S'_2; s_2$. We first show that, after the full execution of $S'_1$ and $S'_2$ started in state $m_1$ and $m_2$, the predicate expression of $s_1$ and $s_2$ evaluate to the same value, w.l.o.g., zero. Next we show that $S_1$ and $S'_1; S'_1$ compute the variable $x$ equivalently when (1) both started in state $m_1$ and (2) the predicate expression of $s_1$ evaluates to zero after the full execution of $S'_1$ started in state $m_1$, similarly $S_2$ and $S'_2; S'_2$ compute the variable $x$ equivalently when (1) both started in the state $m_2$ and (2) the predicate expression of $s_2$ evaluates to zero after the full execution of $S'_2$ when started in state $m_2$. Last we prove the theorem by showing that $S'_1; S'_1$ started in state $m_1$ and $S'_2; S'_2$ started in state $m_2$ compute the variable $x$ equivalently.

In order to show that $S'_1$ and $S'_2$ compute the variables used in predicate expression of $s_1$ and $s_2$ equivalently by the hypothesis IH, we show that all required conditions are satisfied for the application of hypothesis IH.

• size($S'_1$) + size($S'_2$) $< k$.
  The sum of program size of $S'_1$ and $S'_2$ are less than $k$ by the definition of program size for $s_1$ and $s_2$, size($S'_1$) + size($S'_2$) $< k$.

• the execution of $S'_1$ and $S'_2$ terminate, $(S'_1, m_1) \to (\text{skip}, m'_1(\sigma''_1))$, and $(S'_2, m_2) \to (\text{skip}, m'_2(\sigma''_2))$.
  By assumption, the execution of $S_1$ and $S_2$ from the state $m_1$ and $m_2$ respectively terminate, then the execution of $S'_1$ and $S'_2$ terminate when started in state $m_1$ and $m_2$ respectively.

• the initial value stores $\sigma_1$ and $\sigma_2$ agree on the values of the imported variables in $S'_1$ and $S'_2$ relative to the variables used in the predicate expression of $s_1$ and $s_2$.
  By Lemma 4.3.3, the imported variables in $S'_1$ and $S'_2$ relative to the variables used in predicate expression of $s_1$ and $s_2$ are same, Imp($S'_1$, Use($e$)) = Imp($S'_2$, Use($e$)) = Imp($e$). By the definition of imported variable, the imported variables in $S'_1$ relative to the variables used in predicate expression of $s_1$ are a subset of the imported variables in $S_1$ relative to $x$ respectively, Imp($S'_1$, Use($e$)) $\subseteq$ Imp($S'_1$, Imp($s_1$, $\{x\}$)) = Imp($S_1$, $\{x\}$). Similarly Imp($S'_2$, Use($e$)) $\subseteq$ Imp($S_2$, $\{x\}$). Then, by assumption, the initial value stores agree on the values of the imported variables in $S'_1$ and $S'_2$ relative to the variables used in the predicate expression of $s_1$ and $s_2$, $\sigma_1(y) = \sigma_2(y), \forall y \in \text{Imp}(e) = \text{Imp}(S'_1, \text{Use}(e)) = \text{Imp}(S'_2, \text{Use}(e))$.

By the hypothesis IH, after the full execution of $S'_1$ and $S'_2$, the value stores agree on the values of the variables used in the predicate expression of $s_1$ and $s_2$, $\sigma'_1(y) = \sigma''_1(y), \forall y \in \text{Use}(e)$. By Corollary B.3.1, $s_1$ and $s_2$ continue execution after the full execution of $S'_1$ and $S'_2$ respectively, $(S'_1; s_1, m_1) \to (s_1, m'_1(\sigma'_1))$, and $(S'_2; s_2, m_2) \to (s_2, m'_2(\sigma''_2))$. 

180
By the property of expression meaning function $E$, expression $e$ evaluates to the same value w.r.t value stores $\sigma_1'$ and $\sigma_2''$, w.l.o.g., zero, $E[e]_{\sigma_1'} = E[e]_{\sigma_2''} = 0$. Then the execution of $s_1$ proceeds as follows.

$$(s_1, m''_1(\sigma''_1))$$

$$= (\text{If } (e) \text{ then } \{S'_1\} \text{ else } \{S'_1\}, m''_1(\sigma''_1))$$

$$\rightarrow (\text{If } (0) \text{ then } \{S'_1\} \text{ else } \{S'_1\}, m''_1(\sigma''_1))$$ by the EEval rule.

Similarly, the execution from $(s_2, m''_2(\sigma''_2))$ gets to $(S''_2, m''_2(\sigma''_2))$.

Then, we show that $S_1$ and $S'_1; S'_1$ compute the variable $x$ equivalently when both started from state $m_1(\sigma_1)$. The execution of $S'_1; S'_1$ started from state $m_1$ also gets to configuration $(S'_1, m''_1(\sigma''_1))$ because execution of $S_1 = S'_1; s_1$ and $S'_1; S'_1$ share the common execution $(S'_1, m_1) \rightarrow \text{(skip, } m''_1(\sigma''_1))$.

By Corollary B.3.1, $S'_1$ continues execution after the full execution of $S'_1$,

$$(S'_1; S'_1, m_1) \rightarrow (S''_1, m''_1).$$

Therefore, the execution of $S_1$ and $S'_1; S'_1$ from state $m_1$ compute the variable $x$ equivalently because both executions get to same intermediate configuration. Similarly, $S_2$ and $S'_2; S'_2$ compute the variable $x$ equivalently when both started from state $m_2(\sigma_2)$.

Lastly, we show that $S'_1; S'_1$ and $S'_2; S'_2$ compute the variable $x$ equivalently when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively by the hypothesis IH. To do that, we show that all required conditions are satisfied for the application of hypothesis IH.

- $\text{size}(S'_1; S'_1) + \text{size}(S'_2; S'_2) < k$.
  This is obtained by the definition of program size.

- $\text{execution of } S'_1; S'_1 \text{ and } S'_2; S'_2 \text{ terminate when started in state } m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively.
  This is obtained by above argument.

- $\sigma_1(y) = \sigma_2(y), \forall y \in \text{Imp}(S'_1; S'_1, \{x\}) \cup \text{Imp}(S'_2; S'_2, \{x\})$.
  We show that $\text{Imp}(S'_1; S'_1, \{x\}) \subseteq \text{Imp}(S_1, \{x\})$ as follows.

  $$\text{Imp}(S'_1, \{x\}) \subseteq \text{Imp}(s_1, \{x\}) \text{ by the definition of imported variables.}$$

  $$= \text{Imp}(S'_1, \text{Imp}(S'_1, \{x\})) \text{ by Lemma B.1.1}$$

  $$\subseteq \text{Imp}(S'_1, \text{Imp}(s_1, \{x\}) \text{ by (1)})$$

  $$= \text{Imp}(S_1, \{x\}) \text{ by the definition of imported variables.}$$

Similarly, $\text{Imp}(S'_2; S'_2, \{x\}) \subseteq \text{Imp}(S_2, \{x\})$. Then, by assumption, the initial value stores agree on the values of the imported variables in $S'_1; S'_1$ and $S'_2; S'_2$ relative to $x$.  

181
Then, by the hypothesis IH, after the full execution of $S'_1; S'_1$ and $S'_2; S'_2$, the value stores agree on the value of $x$, $(S'_1; S'_1, m_1) \rightarrow^* (\text{skip}, m'_1(\sigma'_1))$, $(S'_2; S'_2, m_2) \rightarrow^* (\text{skip}, m'_2(\sigma'_2))$ such that $\sigma'_1(x) = \sigma'_2(x)$.

In conclusion, after execution of $S_1$ and $S_2$, the value stores agree on the value of $x$. Therefore, the theorem holds.
Theorem 3: If two simple statements $s_1$ and $s_2$ satisfy the proof rule of termination in the same way, $s_1 \equiv^*_H s_2$, and their initial states $m_1(f_1, \sigma_1)$ and $m_2(f_2, \sigma_2)$ with crash flags not set, $f_1 = f_2 = 0$, and whose value stores agree on values of the termination deciding variables of $s_1$ and $s_2$, $\forall x \in \text{TVar}(s_1) \cup \text{TVar}(s_2) : \sigma_1(x) = \sigma_2(x)$, when executions of $s_1$ and $s_2$ start in states $m_1$ and $m_2$ respectively, then $s_1$ and $s_2$ terminate in the same way when started in states $m_1$ and $m_2$ respectively: $(s_1, m_1) \equiv_H (s_2, m_2)$.

Proof.

The proof is a case analysis of those cases in the definition of $s_1 \equiv^*_H s_2$. Because $s_1$ is a simple statement and $s_1$’s execution is without function call, we only care the crash variables of $s_1$ in the termination deciding variables of $s_1$, $\text{CVar}(s_1)$. Similarly, we only care $\text{CVar}(s_2)$.

(First) $s_1$ and $s_2$ are same: $s_1 = s_2$;

We show the theorem by induction on abstract syntax of $s_1$ and $s_2$.

1. $s_1 = s_2 = \text{skip}$.

By definition of termination in the same way, both $s_1$ and $s_2$ terminate. The theorem holds.

2. $s_1 = s_2 = \text{"lval := e"}$.

There are further cases regarding what $lval$ is.

(a) $lval = \text{id}$.

By definition, $\text{CVar}(s_1) = \text{CVar}(s_2) = \text{Err}(e)$ or $\text{Use}(e)$ based on if there is possible value mismatch (e.g., assigning value defined only in type Long to a variable of type Int). There are two subcases.

- Left value $\text{id}$ is of type Int and expression $e$ is of type Long but not type Int, $(\Gamma \vdash \text{id} : \text{Int}) \land (\Gamma \vdash e : \text{Long}) \land \neg(\Gamma \vdash e : \text{Int})$.

By definition, $\text{CVar}(s_1) = \text{CVar}(s_2) = \text{Use}(e)$. By assumption, $\forall x \in \text{Use}(e), \sigma_1(x) = \sigma_2(x)$. By Lemma B.2.1, the expression evaluates to the same value w.r.t two pairs of value stores $\sigma_1$ and $\sigma_2$ respectively,

- Both evaluations of expression lead to crash, $\mathcal{E}[e] \sigma_1 = \mathcal{E}[e] \sigma_2 = (\text{error}, v_{\text{ef}})$.

Then the execution of $s_1$ is as follows:

$$
(s_1, m_1) = (\text{id} := e, m_1(\sigma_1))
$$

$\rightarrow (\text{id} := (\text{error}, \ast), m_1(\sigma_1))$ by rule EEval’

$\rightarrow (\text{id} := 0, m_1(1/f))$ by rule ECrack.

$\rightarrow (\text{id} := 0, m_1(1/f))$ for any $i > 0$ by rule Crash.

Similarly, $s_2$ does not terminate. The theorem holds.

- Both evaluations of expression lead to no crash, $\mathcal{E}[e] \sigma_1 = \mathcal{E}[e] \sigma_2 = (v, v_{\text{ef}})$.

Then there are cases regarding if value mismatch occurs.

- The value $v$ is only defined in type Long, $(\Gamma \vdash v : \text{Long}) \land \neg(\Gamma \vdash v : \text{Int})$.

The execution of $s_1$ is as follows:
$$(s_1, m_1) = (id := e, m_1(\sigma_1))$$

$$\rightarrow (id := (v, v_0), m_1(\sigma_1))$$ by rule EVal

$$\rightarrow (id := v, m_1(\sigma_1))$$ by rule EOflow-1 or EOflow-2.

$$\rightarrow (id := v, m_1(1/f))$$ by rule Assign-Err.

$$i \rightarrow (id := v, m_1(1/f))$$ for any $i > 0$ by rule Crash.

Similarly, $s_2$ does not terminate. The theorem holds.

The value $v$ is defined in type Int, $\Gamma \vdash v : Int$.

Assuming that the variable $id$ is a global one, the execution of $s_1$ is as follows:

$$(s_1, m_1) = (id := e, m_1(\sigma_1))$$

$$\rightarrow (id := (v, v_0), m_1(\sigma_1))$$ by rule EVal

$$\rightarrow (id := v, m_1(\sigma_1))$$ by rule EOflow-1 or EOflow-2.

$$\rightarrow (\text{skip}, m_1(\sigma_1[v/id]))$$ by rule Assign.

Similarly, $s_2$ terminate. The theorem holds.

When the variable $id$ is a local variable, by similar argument for the global variable, we can show that $s_1$ and $s_2$ terminate. Then the theorem holds.

- It is not the case that left value $id$ is of type Int and the expression $e$ is of type Long only,

$$\neg((\Gamma \vdash id : \text{Int}) \land (\Gamma \vdash e : \text{Long}) \land \neg(\Gamma \vdash e : \text{Int})).$$

There are two cases based on if there is crash in evaluation of expression $e$.

√ Both evaluations of expression lead to crash,

$$\mathcal{E}[e]_{\sigma_1} = \mathcal{E}[e]_{\sigma_2} = (\text{error}, v_0).$$

By the same argument in case where left value $id$ is of type Int and the expression $e$ is of type Long only, this theorem holds.

√ Both evaluations of expression lead to no crash,

$$\mathcal{E}[e]_{\sigma_1} = \mathcal{E}[e]_{\sigma_2} = (v, v_0).$$

By the same argument in subcase of no value mismatch in case where left value $id$ is of type Int and the expression $e$ is of type Long only, this theorem holds.

(b) $lvval = id[n]$.

There are two subcases based on if $n$ is within the array bound of $id$. By our assumption, array variable $id$ is of the same bound in two programs. W.l.o.g., we assume $id$ is local variable.

i. $n$ is out of bound of array variable $id$, $((id, n) \mapsto v_1) \notin \sigma_1$ and $((id, n) \mapsto v_2) \notin \sigma_2$;

Then the execution of $s_1$ continues as follows:

$$(s_1, m_1) = (id[n] := e, m_1(\sigma_1))$$

$$\rightarrow (id[n] := e, m_1(1/f))$$ by rule Arr-3

$$\rightarrow (id[n] := e, m_1(1/f))$$ by rule Crash.

Similarly, $s_2$ does not terminate. The theorem holds.
ii. $n$ is within the bound of array variable $id$, $((id, n) \mapsto v_1) \in \sigma_1$ and $((id, n) \mapsto v_2) \in \sigma_2$

There are cases of $\text{CVar}(s_1)$ and $\text{CVar}(s_2)$ based on if there is possible value mismatch exception in $s_1$ and $s_2$.

- Left value $id[n]$ is of type Int and expression $e$ is of type Long but not type Int, $(\Gamma \vdash id[n] : \text{Int}) \land (\Gamma \vdash e : \text{Long}) \land \neg(\Gamma \vdash e : \text{Int})$.

By definition, $\text{CVar}(s_1) = \text{CVar}(s_2) = \text{Use}(e)$. By assumption, $\forall x \in \text{Use}(e), \sigma_1(x) = \sigma_2(x)$. By Lemma B.2.1, the expression evaluates to the same value w.r.t two value stores $\sigma_1$ and $\sigma_2$ respectively,

- Both evaluations of expression lead to crash,

$$\mathcal{E}[e]\sigma_1 = \mathcal{E}[e]\sigma_2 = (\text{error}, v_{st}).$$

Then the execution of $s_1$ is as follows:

\[
\begin{align*}
(s_1, m_1) &= (id[n] := e, m_1(\sigma_1)) \\
\Rightarrow (id[n] &:= (\text{error}, \ast), m_1(\sigma_1)) \text{ by rule } \text{EVal}' \\
\Rightarrow (id[n] &:= 0, m_1(1/f)) \text{ by rule } \text{E Crash}. \\
\Rightarrow (id[n] &:= 0, m_1(1/f)) \text{ for any } i > 0 \\
\text{by rule } \text{Crash}.
\end{align*}
\]

Similarly, $s_2$ does not terminate. The theorem holds.

- Both evaluations of expression lead to no crash, $\mathcal{E}[e]\sigma_1 = \mathcal{E}[e]\sigma_2 = (v, v_{st})$.

Then there are cases regarding if value mismatch occurs.

- The value $v$ is only defined in type Long, $(\Gamma \vdash v : \text{Long}) \land \neg(\Gamma \vdash v : \text{Int})$.

The execution of $s_1$ is as follows:

\[
\begin{align*}
(s_1, m_1) &= (id[n] := e, m_1(\sigma_1)) \\
\Rightarrow (id[n] &:= (v, v_{st}), m_1(\sigma_1)) \text{ by rule } \text{EVal}' \\
\Rightarrow (id[n] &:= v, m_1(\sigma_1)) \\
\text{by rule } \text{EOflow-1 or EOflow-2.} \\
\Rightarrow (id[n] &:= v, m_1(1/f)) \text{ by rule } \text{Assign-Err.} \\
\Rightarrow (id[n] &:= v, m_1(1/f)) \text{ for any } i > 0 \\
\text{by rule } \text{Crash}.
\end{align*}
\]

Similarly, $s_2$ does not terminate. The theorem holds.

- The value $v$ is defined in type Int, $(\Gamma \vdash v : \text{Int})$.

The execution of $s_1$ is as follows:

\[
\begin{align*}
(s_1, m_1) &= (id[n] := e, m_1(\sigma_1)) \\
\Rightarrow (id[n] &:= (v, v_{st}), m_1(\sigma_1)) \text{ by rule } \text{EVal}' \\
\Rightarrow (id[n] &:= v, m_1(\sigma_1)) \\
\text{by rule } \text{EOflow-1 or EOflow-2.} \\
\Rightarrow (id[n] &:= v, m_1(1/f)) \text{ by rule } \text{Assign-A.} \\
\Rightarrow (\text{skip}, m_1(\sigma_1[v/(id, n)])) \text{ by rule } \text{Assign-A.} \\
\end{align*}
\]

Similarly, $s_2$ terminate. The theorem holds.
When the variable $id$ is a global variable, by similar argument for the global variable, we can show that $s_1$ and $s_2$ terminate. Then the theorem holds.

- It is not the case that left value $id$ is of type Int and the expression $e$ is of type Long only,
  \[ \neg((\Gamma \vdash id : \text{Int}) \land (\Gamma \vdash e : \text{Long}) \land \neg(\Gamma \vdash e : \text{Int})). \]
  There are two cases based on if there is crash in evaluation of expression $e$.
  - Both evaluations of expression lead to crash,
    \[ \mathcal{E}[e]\sigma_1 = \mathcal{E}[e]\sigma_2 = (\text{error}, v_{\text{err}}). \]
    By the same argument in case where left value $id$ is of type Int and the expression $e$ is of type Long only, this theorem holds.
  - Both evaluations of expression lead to no crash,
    \[ \mathcal{E}[e]\sigma_1 = \mathcal{E}[e]\sigma_2 = (v, v_{\text{err}}). \]
    By the same argument in subcase of no value mismatch in case where left value $id$ is of type Int and the expression $e$ is of type Long only, this theorem holds.

If array variable $id$ is a global variable, by similar argument above, the theorem holds.

(c) $lval = id_1[id_2]$.
By definition, $\text{Idx}(s_1) = \text{Idx}(s_2) = \{id_2\} \subseteq \text{CVar}(s_1) = \text{CVar}(s_2)$. By assumption, $\sigma_1(id_2) = \sigma_2(id_2) = n$. By the same argument in the case where $lval = id[n]$, the theorem holds.

3. $s_1 = s_2 = \text{"input id"}$.
By definition, $\text{CVar}(s_1) = \text{CVar}(s_2) = \{id_I\}$. By assumption $\sigma_1(id_I) = \sigma_2(id_I)$. There are cases regarding if input sequence is empty or not.

(a) There is empty input sequence, $\sigma_1(id_I) = \sigma_2(id_I) = \emptyset$.
Then the execution of $s_1$ continues as follows:
\[
(s_1, m_1) = (\text{input id}, m_1(\sigma_1))
\rightarrow (\text{input id}, m_1(1/f)) \text{ by rule In-7}
\rightarrow (\text{input id}, m_1(1/f)) \text{ by rule Crash.}
\]
Similarly, $s_2$ does not terminate. The theorem holds.

(b) There is nonempty input sequence, $\sigma_1(id_I) = \sigma_2(id_I) \neq \emptyset$.
There are cases regarding if type of the variable $id$ is Long or not.

i. $id$ is of type Long, $\Gamma \vdash id : \text{Long}$;
Assuming $id$ is a local variable, then the execution of $s_1$ continues as follows:
\[
(s_1, m_1) = (\text{input id}, m_1(\sigma_1))
\rightarrow (\text{skip}, m_1(\sigma_1[v_{\text{io}}/id, \text{tl}(\sigma_1(id_I))/id_I,
\text{"}\sigma_1(id_{IO}) \cdot v_{\text{io}}/id_{IO})]) \text{ by rule In-3.}
\]
Similarly, $s_2$ terminates. The theorem holds.
When the variable $id$ is a global variable, by similar argument, the theorem holds.

ii. $id$ is of type Int or enumeration, $\Gamma \vdash id : \text{Int or enum } id'$;

There are cases regarding if the head of input sequence can be transformed to type of $id$. Let $v_{io} = \text{hd}(\sigma_1(id_I))$.

- $id$ is of type Int.
  If $v_{io}$ is not of type Int, $\Gamma \vdash v_{io} : \text{Long and } \neg (\Gamma \vdash v_{io} : \text{Int})$, then the execution of $s_1$ continues as follows:
    
    $$(s_1, m_1) = (\text{input } id, m_1(\sigma_1))$$
    $$(\text{input } id, m_1(1/f)) \text{ by Rule In-4.}$$
    $$\overset{i}{\rightarrow} (\text{input } id, m_1(1/f)) \text{ by Rule crash.}$$

  Similarly, $s_2$ does not terminate. The theorem holds.

If $v_{io}$ is of type Int, $\Gamma \vdash v_{io} : \text{Long and } \Gamma \vdash v_{io} : \text{Int}$, assuming $id$ is a local variable, then the execution of $s_1$ continues as follows:

$$(s_1, m_1) = (\text{input } id, m_1(\sigma_1))$$

$$\rightarrow (\text{skip, } m_1(\sigma_1[v_{io}/id, \text{tl}(\sigma_1(id_I))/id_I, \text{"} \sigma_1(id_{IO}) \cdot v_{io} /id_{IO}\text{"}])) \text{ by Rule In-8.}$$

Similarly, $s_2$ terminates. The theorem holds.

When $id$ is a global variable, by similar argument, the theorem holds.

- If $id$ is of type enum $id' = \{l_1, ..., l_k\}$.
  If $(v_{io} < 1) \lor (v_{io} > k)$, then the execution of $s_1$ continues as follows:

    $$(s_1, m_1) = (\text{input } id, m_1(\sigma_1))$$
    $$\rightarrow (\text{input } id, m_1(1/f)) \text{ by Rule In-6.}$$
    $$\overset{i}{\rightarrow} (\text{input } id, m_1(1/f)) \text{ by Rule crash.}$$

  Similarly, $s_2$ does not terminate. The theorem holds. When $id$ is a global variable, by similar argument, the theorem holds.

If $1 \leq v_{io} \leq k$, assuming $id$ is a local variable, then the execution of $s_1$ continues as follows:

$$(s_1, m_1) = (\text{input } id, m_1(\sigma_1))$$

$$\rightarrow (\text{skip, } m_1(\sigma_1[l_{v_{io}}/id, \text{tl}(\sigma_1(id_I))/id_I, \text{"} \sigma_1(id_{IO}) \cdot v_{io} /id_{IO}\text{"}])) \text{ by Rule In-5.}$$

Similarly, $s_2$ terminates. The theorem holds. When $id$ is a global variable, by similar argument, the theorem holds.

(c) $s_1 = s_2 = \text{"} \text{output e}\text{"}$;

There are two cases based on if evaluation of expression $e$ crashes. By definition, $CVar(s_1) = CVar(s_2) = Err(e)$. By assumption, $\forall x \in \text{Err(e)}, \sigma_1(x) = \sigma_2(x)$. By Lemma B.2.2, evaluation of the expression $e$ w.r.t two value stores $\sigma_1$ and $\sigma_2$ either both crash or both do not crash.
i. There is crash in evaluation of the expression \( e \) w.r.t two value stores \( \sigma_1 \) and \( \sigma_2 \), \( \mathcal{E}[e] \sigma_1 = (\text{error}, v_{\sigma_1}^1) \) and \( \mathcal{E}[e] \sigma_2 = (\text{error}, v_{\sigma_2}^2) \). The execution of \( s_1 \) continues as follows:

\[
(s_1, m_1) \\
= (\text{output } e, m_1(\sigma_1)) \\
\rightarrow (\text{output } (\text{error}, v_{\sigma_1}^1), m_1(1/f)) \text{ by Rule EEval}' \\
\rightarrow (\text{output } 0, m_1(1/f)) \text{ by Rule ECrash.} \\
\rightarrow (\text{output } 0, m_1(1/f)) \text{ by Rule crash.}
\]

Similarly, \( s_2 \) does not terminate. The theorem holds.

ii. There is no crash in evaluation of the expression \( e \) w.r.t two value stores \( \sigma_1 \) and \( \sigma_2 \), \( \mathcal{E}[e] \sigma_1 = (v_1, v_{\sigma_1}^1) \) and \( \mathcal{E}[e] \sigma_2 = (v_2, v_{\sigma_2}^2) \). According to rule Out-1 and Out-2, there is no exception in transformation of different typed output value. We therefore only show the execution for output value of Int type. The execution of \( s_1 \) continues as follows:

\[
(s_1, m_1) \\
= (\text{output } e, m_1(\sigma_1)) \\
\rightarrow (\text{output } (v_1, v_{\sigma_1}^1), m_1(1/f)) \text{ by Rule EEval’} \\
\rightarrow (\text{output } v_1, m_1(v_{\sigma_1}^1/f)) \text{ by Rule EOflow-1 or EOflow-2.} \\
\rightarrow (\text{skip}, m_1(\sigma_1["\sigma(id_{IO}) \cdot \overline{v_1}"/id_{IO}])) \text{ by Rule Out-1.}
\]

Similarly, \( s_2 \) terminates. Theorem holds.

(Second) \( s_1 \) and \( s_2 \) are input statement with same type variable: \( s_1 = "\text{input } id_1" \), \( s_2 = "\text{input } id_2" \) where \( (\Gamma_{s_1} \vdash id_1 : t) \land (\Gamma_{s_2} \vdash id_2 : t) \).

The theorem holds by similar argument for the case \( s_1 = s_2 = \text{input } id \).

(Third) \( s_1 = "\text{output } e" \) or "\( id_1 := e \)" , \( s_2 = "\text{output } e" \) or "\( id_2 := e \)" where both of the following hold:

- There is no possible value mismatch in "\( id_1 := e \)" , \( \neg(\Gamma_{s_1} \vdash id_1 : \text{Int}) \lor \neg(\Gamma_{s_1} \vdash e : \text{Long}) \lor (\Gamma_{s_1} \vdash e : \text{Int}) \).
- There is no possible value mismatch in "\( id_2 := e \)" , \( \neg(\Gamma_{s_2} \vdash id_2 : \text{Int}) \lor \neg(\Gamma_{s_2} \vdash e : \text{Long}) \lor (\Gamma_{s_2} \vdash e : \text{Int}) \).

We show that the evaluations of the expression \( e \) w.r.t the value stores \( \sigma_1 \) and \( \sigma_2 \) either both raise an exception or both do not. By the definition of crash variables, the crash variables of \( s_1 \) are those obtained by the function \( \text{Err}(e), \text{CVar}(s_1) = \text{Err}(e) \). Similarly, the termination deciding variables of \( s_2 \) are \( \text{Err}(e) \). By assumption, the initial value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of those in \( \text{CVar}(s_1) \) and \( \text{CVar}(s_2) \), \( \forall x \in \text{Err}(e) = (\text{CVar}(s_1) \cup \text{CVar}(s_2)) : \sigma_1(x) = \sigma_2(x) \). By Lemma B.2.2, the evaluations of expression \( e \) w.r.t two value stores, \( \sigma_1 \) and \( \sigma_2 \), either both raise an exception or both do not raise an exception.
1. The evaluations of the expression $e$ raise an exception w.r.t two value stores $\sigma_1$ and $\sigma_2$, $\mathcal{E}[e] \sigma_1 = (\text{error}, v^1_{\sigma_1}), \mathcal{E}[e] \sigma_2 = (\text{error}, v^2_{\sigma_1})$.

We show the execution of $s_1$ proceeds to a configuration where the crash flag is set and then does not terminate.

When $s_1 = \text{"output } e\text{"}$, the execution of “output $e$” proceeds as follows.

\[
\begin{align*}
&\text{(output } e, m_1(\sigma_1)) \\
&\quad \rightarrow \text{(output (error, $v^1_{\sigma_1}$), } m_1(\sigma_1)) \text{ by rule EEval'} \\
&\quad \rightarrow \text{(output 0, } m_1(1/f)) \text{ by rule ECrash} \\
&\quad \rightarrow^i \text{(output 0, } m_1(1/f)) \text{ for any } i \geq 0, \text{ by rule Crash.}
\end{align*}
\]

When $s_1 = \text{"id}_1 := e\text{"}$, the execution of “$\text{id}_1 := e$” proceeds as follows.

\[
\begin{align*}
&\text{(id}_1 := e, m_1(\sigma_1)) \\
&\quad \rightarrow \text{(id}_1 := (error, $v^1_{\sigma_1}$), } m_1(\sigma_1)) \text{ by rule EEval'} \\
&\quad \rightarrow \text{(id}_1 := 0, } m_1(1/f)) \text{ by rule ECrash} \\
&\quad \rightarrow^i \text{(id}_1 := 0, } m_1(1/f)) \text{ for any } i \geq 0, \text{ by rule Crash.}
\end{align*}
\]

Similarly, the execution of $s_2$ proceeds to a configuration where the crash flag is set. Then, by the crash rule, the execution of $s_2$ does not terminate.

The theorem 3 holds.

2. the evaluations of expression $e$ do not raise an exception w.r.t two value stores, $\sigma_1$ and $\sigma_2$, $\mathcal{E}[e] \sigma_1 = (v_1, v^1_{\sigma_1}), \mathcal{E}[e] \sigma_2 = (v_2, v^2_{\sigma_1})$.

We show the execution of $s_1$ terminates.

When $s_1 = \text{output } (e)$, the execution of output $(e)$ proceeds as follows. W.l.o.g, we assume expression $e$ is of type Int. This is allowed by the condition that it does not hold that $(\Gamma_{s_1} \vdash e : \text{Long}) \land \neg(\Gamma_{s_1} \vdash e : \text{Int})$.

\[
\begin{align*}
&\text{(output } e, m_1(\sigma_1)) \\
&\quad \rightarrow \text{(output } v_1, v^1_{\sigma_1}, m_1(\sigma_1)) \text{ by rule EEval'} \\
&\quad \rightarrow \text{(output } v_1, m_1(v^1_{\sigma_1/\sigma_1}, \sigma_1)) \text{ by rule E-Oflow1 or E-Oflow2} \\
&\quad \rightarrow \text{(skip, } m_1(\sigma_1[\text{"id}_1 : \text{id}_0\cdot v_1" / \text{id}_1])) \text{ by rule Out.}
\end{align*}
\]

When $s_1 = \text{"id}_1 := e\text{"}$, by assumption, the expression $e$ is of type Int, there is no possible value mismatch in execution of “$\text{id}_1 := e$” because the only possible value mismatch occurs when assigning a value of type Long but not Int to a variable of type Int. By the condition $\neg(\Gamma_{s_1} \vdash \text{id}_1 : \text{Int}) \lor \neg(\Gamma_{s_1} \vdash e : \text{Long}) \lor (\Gamma_{s_2} \vdash e : \text{Int})$, when expression $e$ is of type Long, then the variable $\text{id}_1$ is not of type Int. In summary, there is no value mismatch.

The execution of “$\text{id}_1 := e$” proceeds as follows.

\[
\begin{align*}
&\text{(id}_1 := e, m_1(\sigma_1)) \\
&\quad \rightarrow \text{(id}_1 := (v_1, v^1_{\sigma_1}), m_1(\sigma_1)) \text{ by rule EEval'} \\
&\quad \rightarrow \text{(id}_1 := v_1, m_1(v^1_{\sigma_1/\sigma_1}, \sigma_1)) \text{ by rule EEval'} \\
&\quad \rightarrow \text{(skip, } m_1(\sigma_1[v_1/\text{id}_1])) \text{ by the rule Assign.}
\end{align*}
\]
When $id_1$ is a variable of enumeration or Long type, by similar argument, the theorem still holds.

Similarly, the execution of $s_2$ terminates when started in the state $m_2(s_2)$. Theorem 3 holds.

Theorem 4: If two statement sequences $S_1$ and $S_2$ satisfy the proof rule of termination in the same way, $S_1 \equiv^s_H S_2$, and their respective initial states $m_1(f_1, \sigma_1)$ and $m_2(f_2, \sigma_2)$ with crash flags not set, $f_1 = f_2 = 0$, and whose value stores agree on values of the termination deciding variables of $S_1$ and $S_2$, $\forall x \in \text{TVar}(S_1) \cup \text{TVar}(S_2): \sigma_1(x) = \sigma_2(x)$, then $S_1$ and $S_2$ terminate in the same way when started in states $m_1$ and $m_2$ respectively: $(S_1, m_1) \equiv^s_H (S_2, m_2)$.

Proof.

The proof is by induction on $\text{size}(S_1) + \text{size}(S_2)$, the sum of program size of $S_1$ and $S_2$.

Base case. $S_1$ and $S_2$ are simple statement. By Theorem 3, Theorem 4 holds.

Induction step.

There are two hypotheses. The hypothesis IH is that Theorem 4 holds when $\text{size}(S_1) + \text{size}(S_2) = k \geq 2$.

We show Theorem 4 holds when $\text{size}(S_1) + \text{size}(S_2) = k + 1$.

The proof of Theorem 4 is a case analysis according to the cases in the definition of the proof rule of termination in the same way, $S_1 \equiv^s_H S_2$.

1. $S_1$ and $S_2$ are one statement and one of the following holds.

   (a) $S_1 = \text{"If}(\epsilon) \text{ then } \{S_1^f\} \text{ else } \{S_1^f\}$, $S_2 = \text{"If}(\epsilon) \text{ then } \{S_2^f\} \text{ else } \{S_2^f\}$ such that one of the following holds:

   i. $S_1^f, S_2^f, S_1^c, S_2^c$ are all sequences of “skip”;

   We show that the evaluation of expression $\epsilon$ w.r.t the value store $\sigma_1$ and $\sigma_2$ either both raise an exception or both do not. By the definition of crash/loop variables, $\text{CVar}(S_1^c) = \text{CVar}(S_2^c) = \emptyset$, $\text{LVar}(S_1) = \emptyset$. By the definition of termination deciding variables, the termination deciding variables of $S_1$ is the crash variables of $S_1$, $\text{TVar}(S_1) = \text{CVar}(S_1) = \text{Err}(\epsilon)$. By assumption, the value stores $\sigma_1$ and $\sigma_2$ agree on the values of those in the crash variables of $S_1$ and $S_2$, $\forall x \in \text{Err}(\epsilon) = \text{TVar}(S_1) = \text{TVar}(S_2), \sigma_1(x) = \sigma_2(x)$. By the property of the expression meaning function $E$, the evaluation of predicate expression $\epsilon$ of $S_1$ and $S_2$ w.r.t value store $\sigma_1$ and $\sigma_2$ either both crash or both do not crash, $(E[\epsilon]_1 \sigma_1 = E[\epsilon]_2 \sigma_2 = \text{error}) \lor ((E[\epsilon]_1 \sigma_1 \neq \text{error}) \land (E[\epsilon]_2 \sigma_2 \neq \text{error}))$.

   Then we show that Theorem 4 holds in either of the two possibilities.

   A. $E[\epsilon]_1 \sigma_1 = E[\epsilon]_2 \sigma_2 = \text{error}$.

   The execution of $S_1$ proceeds as follows:

   $\text{If}(\epsilon) \text{ then } \{S_1^f\} \text{ else } \{S_1^f\}, m_1(\sigma_1)$

   $\rightarrow \text{If(error) then } \{S_1^f\} \text{ else } \{S_1^f\}, m_1(\sigma_1) \text{ by rule EEval}$

   $\rightarrow \text{If(0) then } \{S_1^f\} \text{ else } \{S_1^f\}, m_1(1/\sigma_1) \text{ by rule ECrash}$

   $i \rightarrow \text{If(0) then } \{S_1^f\} \text{ else } \{S_1^f\}, m_1(1/\sigma_1) \text{ for any } i \geq 0,$

   by rule Crash.
Similarly, the execution of $S_2$ started in the state $m_2(\sigma_2)$ does not terminate. The theorem 4 holds.

B. $(\mathcal{E}[e]\sigma_1 \neq \text{error}) \land (\mathcal{E}[e]\sigma_2 \neq \text{error})$.

W.l.o.g. $\mathcal{E}[e]\sigma_1 = v_1 \neq 0$, $\mathcal{E}[e]\sigma_2 = 0$. Then the execution of $S_1$ proceeds as follows.

- if (e) then $\{S'_1\}$ else $\{S'_1, m_1(\sigma_1)\}$

- $\rightarrow$(if (v1) then $\{S'_1\}$ else $\{S'_1, m_1(\sigma_1)\}$) by rule EEval

- $\rightarrow$(if (f) then $\{S'_1\}$ else $\{S'_1, m_1(\sigma_1)\}$) by rule If-T

Similarly, the execution of $S_2$ started in the state $m_2(\sigma_2)$ terminates. The theorem 4 holds.

ii. At least one of $S'_1, S'_2$ is not a sequence of “skip” and $(S'_1 \equiv^S S'_2) \land (S'_2 \equiv^S S'_2)$.

W.l.o.g., $S'_1$ is not of “skip” only. We show that the evaluation of the expression $e$ w.r.t the value stores $\sigma_1$ and $\sigma_2$ either both raise an exception or both produce the same integer value. Then there is either some loop statement in $S'_1$ or the crash variables of $S'_1$ are not $\emptyset$ or both.

A. When there is some loop statement in $S'_1$, then, by the definition of loop variables, the loop variables of $S_1$ include all variables used in the predicate expression of $S_1$, $\text{LVar}(S_1) = \text{Use}(e) \cup \text{LVar}(S'_1) \cup \text{LVar}(S'_1)$.

B. When the crash variables of $S'_1$ are not $\emptyset$, then, by the definition of crash variables, the crash variables of $S_1$ include all variables used in the predicate expression of $S_1$, $\text{CVar}(S_1) = \text{Use}(e) \cup \text{CVar}(S'_1) \cup \text{CVar}(S'_1)$.

In summary, all variables used in predicate expression of $S_1$ is a subset of termination deciding variables of $S_1$, $\text{Use}(e) \subseteq \text{TVar}(S_1)$. By assumption, the value store $\sigma_1$ and $\sigma_2$ agree on the values of those in the termination deciding variables of $S_1$ and $S_2$. It follows, by the property of expression meaning function $\mathcal{E}$, the evaluation of the predicate expression $e$ of $S_1$ and $S_2$ produce the same value w.r.t the value store $\sigma_1$ and $\sigma_2$, $\mathcal{E}[e]\sigma_1 = \mathcal{E}[e]\sigma_2$. Then either the evaluations of the predicate expression $e$ of $S_1$ and $S_2$ both crash w.r.t the value store $\sigma_1$ and $\sigma_2$, or both evaluations produce the same integer value, $(\mathcal{E}[e]\sigma_1 = \mathcal{E}[e]\sigma_2 = \text{error}) \lor (\mathcal{E}[e]\sigma_1 = \mathcal{E}[e]\sigma_2 = v \neq \text{error})$. We show Theorem 4 holds in either of the two possibilities.

A. $\mathcal{E}[e]\sigma_1 = \mathcal{E}[e]\sigma_2 = \text{error}$.

The execution of $S_1$ proceeds as follows:

- $(\text{if } e \text{ then } \{S'_1\} \text{ else } \{S'_1, m_1(\sigma_1)\})$

- $\rightarrow$(if (error) then $\{S'_1\}$ else $\{S'_1, m_1(\sigma_1)\}$) by rule EEval

- $\rightarrow$(if (0) then $\{S'_1\}$ else $\{S'_1, m_1(1/f, \sigma_1)\}$) by rule ECrash

- $(\text{if } 0 \text{ then } \{S'_1\} \text{ else } \{S'_1, m_1(1/f, \sigma_1)\})$ for any $i \geq 0$, by rule Crash.
Similarly, the execution of $S_2$ started from state $m_2(\sigma_2)$ does not terminate. The theorem 4 holds.

B. $\mathcal{E}[e]\sigma_1 = \mathcal{E}[e]\sigma_2 = v \neq \text{error}$, w.l.o.g., $v = 0$. Then the execution of $S_1$ proceeds as follows:

- $(\text{If}(e) \text{ then } \{S_1^f\} \text{ else } \{S_2^f\}, m_1(\sigma_1))$
  - $(\text{If}(0) \text{ then } \{S_1^f\} \text{ else } \{S_1^f\}, m_1(\sigma_1))$ by rule EEval
  - $(S_1^f, m_1(\sigma_1))$ by rule If-F.

Similarly, after two steps of execution, $S_2$ gets to the configuration $(S_2^f, m_2(\sigma_2))$.

We show that $S_1^f$ and $S_2^f$ terminate in the same way when started in the state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. Because $S_1^f \equiv_H S_2^f$, by Corollary 4.4.1, the termination deciding variables of $S_1^f$ and $S_2^f$ are same, $\text{TVar}(S_1^f) = \text{TVar}(S_2^f)$. By the definition of crash/loop variables, $\text{CVar}(S_1^f) \subseteq \text{CVar}(S_1)$ and $\text{LVar}(S_1^f) \subseteq \text{LVar}(S_1)$. Hence, the termination deciding variables of $S_1^f$ are a subset of the termination deciding variables of $S_1$, $\text{TVar}(S_1^f) \subseteq \text{TVar}(S_1)$. Similarly, $\text{TVar}(S_2^f) \subseteq \text{TVar}(S_2)$. Therefore, the value store $\sigma_1$ and $\sigma_2$ agree on the values of those in the termination deciding variables of $S_1^f$ and $S_2^f$, $\forall y \in \text{TVar}(S_1^f) \cup \text{TVar}(S_2^f) : \sigma_1(y) = \sigma_2(y)$.

In addition, the sum of program size of $S_1^f$ and $S_2^f$ is less than $k$ because program size of each of $S_1^f$ and $S_2^f$ is greater than or equal to one, $\text{size}(S_1^f) + \text{size}(S_2^f) < k$. As is shown, crash flags are not set. Therefore, by the hypothesis IH, $S_1^f$ and $S_2^f$ terminate in the same way when started in state $m_1(f_1, \sigma_1)$ and $m_2(f_2, \sigma_2)$, $(S_1^f, m_1(f_1, \sigma_1)) \equiv_H (S_2^f, m_2(f_2, \sigma_2))$. Hence, Theorem 4 holds.

(b) $S_1 = \text{“while}_{(n_1)}(e) \{S_1''\}$, $S_2 = \text{“while}_{(n_2)}(e) \{S_2''\}$ such that both of the following hold:

- $S_1'' \equiv_H S_2''$.
- $S_1''$ and $S_2''$ have equivalent computation of $\text{TVar}(S_1) \cup \text{TVar}(S_2)$.

By Corollary 4.4.3, we show $S_1$ and $S_2$ terminate in the same way when started from state $m_1(f_1, m^1_1, \sigma_1)$ and $m_2(f_2, m^2_2, \sigma_2)$ respectively. We need to show that all required conditions are satisfied.

- The crash flags are not set, $f_1 = f_2 = 0$.
- The loop counter value of $S_1$ and $S_2$ are zero: $m^1_1(n_1) = m^2_2(n_2) = 0$.
- The value stores $\sigma_1$ and $\sigma_2$ agree on the values of those in the termination deciding variables of $S_1$ and $S_2$, $\forall x \in \text{TVar}(S_1) \cap \text{TVar}(S_2) : \sigma_1(x) = \sigma_2(x)$. The three above conditions are from assumption.
- $S_1$ and $S_2$ have same set of termination deciding variables, $\text{TVar}(S_1) = \text{TVar}(S_2)$.

By Corollary 4.4.1.
• The loop body $S''_1$ of $S_1$ and $S''_2$ of $S_2$ terminate in the same way when started in state $m_{S_1}(f_1, \sigma_{S_1})$ and $m_{S_2}(f_2, \sigma_{S_2})$ with crash flags not set and in which value stores agree on the values of those in the termination deciding variables of $S''_1$ and $S''_2$: $((\forall x \in \text{TVar}(S''_1) \cup \text{TVar}(S''_2)) : \sigma_{S_1}(x) = \sigma_{S_2}(x)) \land (f_{S_1} = f_{S_2} = 0)$ \Rightarrow (S''_1, m_{S_1}(f_1, \sigma_{S_1})) \equiv_H (S''_2, m_{S_2}(f_2, \sigma_{S_2})).$

By the definition of program size, size($S_1$) = size($S''_1$) + 1, size($S_2$) = size($S''_2$) + 1. Then, size($S''_1$) + size($S''_2$) < k. Then, by the hypothesis IH, the loop body $S''_1$ of $S_1$ and $S''_2$ of $S_2$ terminate in the same way when started in state $m_{S_1}(\sigma_{S_1})$ and $m_{S_2}(\sigma_{S_2})$ with crash flags not set and whose value stores agree on values of the termination deciding variables of $S''_1$ and $S''_2$.

Then, by Corollary 4.4.3, $S_1$ and $S_2$ terminate in the same way when started in the states $m_1(m^1_1, \sigma_1)$ and $m_2(m^2_2, \sigma_2)$ respectively. The theorem 4 holds.

2. $S_1$ and $S_2$ are not both one statement and one of the following holds:

(a) $S_1 = S'_1; s_1, S_2 = S'_2; s_2$ and all of the following hold:

• $S'_1 \equiv_H S'_2$;
• $S_1$ and $S'_2$ have equivalent computation of $\text{TVar}(s_1) \cup \text{TVar}(s_2)$;
• $s_1 \equiv_H s_2$ where $s_1$ and $s_2$ are not sequences of “skip”;

By the hypothesis IH, we show that $S'_1$ and $S'_2$ terminate in the same way when started in the states $m_1(f_1, \sigma_1), m_2(f_2, \sigma_2)$ respectively, $(S'_1, m_1(f_1, \sigma_1)) \equiv_H (S'_2, m_2(f_2, \sigma_2))$. We need to show all required conditions are satisfied.

• Crash flags are not set, $f_1 = f_2 = 0$.
  By assumption.
• size($S'_1$) + size($S'_2$) < k.
  By the definition, size($s_1$) \geq 1, size($s_2$) \geq 1. Hence size($S'_1$)+size($S'_2$) < k.
• Value stores $\sigma_1$ and $\sigma_2$ agree on values of the termination deciding variables of $S'_1$ and $S'_2$.

Besides, by the definition of loop/crash variables, $LVar(S'_1) \subseteq LVar(S_1)$ and $CVar(S'_1) \subseteq CVar(S_1)$. Hence, $\text{TVar}(S'_1) \subseteq \text{TVar}(S_1)$. Similarly, $\text{TVar}(S'_2) \subseteq \text{TVar}(S_2)$. Then, value stores $\sigma_1$ and $\sigma_2$ agree on the values of those in the termination deciding variables of $S'_1$ and $S'_2$, $\forall x \in \text{TVar}(S'_1) \cup \text{TVar}(S'_2) : \sigma_1(x) = \sigma_2(x)$.

Then, by the hypothesis IH, $S'_1$ and $S'_2$ terminate in the same way when started in the states $m_1(f_1, \sigma_1), m_2(f_2, \sigma_2)$ respectively, $(S'_1, m_1(f_1, \sigma_1)) \equiv_H (S'_2, m_2(f_2, \sigma_2))$.

If the execution of $S'_1$ and $S'_2$ terminate when started in the states $m_1(f_1, \sigma_1)$ and $m_2(f_2, \sigma_2)$ respectively, we show that $s_1$ and $s_2$ terminate in the same way. We prove that $S'_1$ and $S'_2$ equivalently compute the termination deciding variables of $s_1$ and $s_2$ by Theorem 2.
• Crash flags are not set, \( f_1 = f_2 = 0 \);
By definition of terminating execution of \( S'_1 \) and \( S'_2 \) when started in states \( m_1 \) and \( m_2 \) respectively.

• The executions of \( S'_1 \) and \( S'_2 \) terminate when started in the states

\[
(\text{size}(m_1((\sigma_1))) = \text{size}(m_2((\sigma_2))) = \text{size}(m_1(\sigma_1)) = \text{size}(m_2(\sigma_2)).
\]

By assumption, \( (S'_1, m_1(\sigma_1)) \xrightarrow{\ast} (s_1, m'_1(\sigma'_1)) \) and \( (S'_2, m_2(\sigma_2)) \xrightarrow{\ast} (skip, m'_2(\sigma'_2)). \)

• \( s_1 \) and \( s_2 \) have same termination deciding variables.
By Corollary 4.4.1, \( s_1 \) and \( s_2 \) have same termination deciding variables, \( \text{TVar}(s_1) = \text{TVar}(s_2) = \text{TVar}(s). \)

• Value stores \( \sigma_1 \) and \( \sigma_2 \) agree on the values of variables in

\( \text{Imp}(S'_1, \text{TVar}(s)) \cup \text{Imp}(S'_2, \text{TVar}(s)). \)

By the definition of loop/crash variables, \( \text{Imp}(S'_1, \text{LVar}(s_1)) \subseteq \text{LVar}(S_1) \) and \( \text{Imp}(S'_1, \text{CVar}(s_1)) \subseteq \text{CVar}(S_1). \) Hence, by Lemma B.1.2, the imported variables in \( S'_1 \) relative to the termination deciding variables of \( s_1 \) is a subset of the termination deciding variables of \( S_1 \).

Thus, by assumption, the value stores \( \sigma_1 \) and \( \sigma_2 \) agree on the values of the variables in \( \text{Imp}(S'_1, \text{TVar}(s)) \cup \text{Imp}(S'_2, \text{TVar}(s)). \)

By Theorem 2, \( \forall x \in \text{TVar}(s) : \sigma'_1(x) = \sigma'_2(x). \)

By Corollary B.3.1, \( (s'_1; s_1, m_1(\sigma_1)) \xrightarrow{\ast} (s_1, m'_1(f_1, \sigma'_1)) \) and \( (s'_2; s_2, m_2(\sigma_2)) \xrightarrow{\ast} (s_2, m'_2(f_2, \sigma'_2)). \) Then, by the hypothesis IH, we show that \( s_1 \) and \( s_2 \) terminate in the same way when started in the states \( m'_1(\sigma'_1) \) and \( m'_2(\sigma'_2). \)

We show that all required conditions are satisfied. \( \text{size}(s_1) + \text{size}(s_2) < k \)
because \( \text{size}(S'_1) \geq 1, \text{size}(S'_2) \geq 1 \) by the definition of program size. If \( s_1, s_2 \) are loop statement, then, by the assumption of unique loop labels,
\( s_1 \notin S'_1, s_2 \notin S'_2. \) Then, by Corollary B.3.4, the loop counter value of \( s_1 \) and \( s_2 \) is not redefined in the execution of \( S'_1 \) and \( S'_2 \) respectively. By the hypothesis IH, \( s_1 \) and \( s_2 \) terminate in the same way when started in the states \( m'_1(f_1, \sigma'_1) \) and \( m'_2(f_2, \sigma'_2), \) \( (s_1, m'_1(f_1, \sigma'_1)) \equiv_H (s_2, m'_2(f_2, \sigma'_2)). \) The theorem 4 holds.

(b) One last statement is “skip”: w.l.o.g., \( (s_2 = \text{“skip”}) \land (S_1 \equiv_H S'_2). \)

We show that \( S_1 \) and \( S'_2 \) terminate in the same way when started in the states \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) respectively by the hypothesis IH. By the definition of crash/loop variables, \( \text{CVar}(S'_2) \subseteq \text{CVar}(S_2), \text{LVar}(S'_2) \subseteq \text{LVar}(S_2). \)

Then, by assumption, \( \forall x \in \text{TVar}(S'_2) \cup \text{TVar}(S_1) : \sigma_1(x) = \sigma_2(x). \)

Besides, size \( (s_2) \geq 1 \) by the definition of program size. Then size \( (S_1) + \text{size} (S'_2) \leq k. \) By the hypothesis IH, \( S_1 \) and \( S'_2 \) terminate in the same way when started in the states \( m_1(f_1, \sigma_1), m_2(f_2, \sigma_2), \) \( (S_1, m_1(f_1, \sigma_1)) \equiv_H (S'_2, m_2(f_2, \sigma_2)). \)

When the execution of \( S_1 \) and \( S'_2 \) terminate when started in the states \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) respectively, \( s_2 \) terminates after the execution of \( S'_2 \) by the definition of terminating execution.

(c) One last statement is a “duplicate” statement such that one of the following holds:
W.l.o.g., \( S_2 = S'_2; S'_2; S''_2; s_2 \) and all of the following hold:
• $S_1 \equiv^S_H S_2$; $s'_2; S''_2$;
• $s'_2 \equiv^S_H s_2$;
• Def($s'_2; S''_2$) \cap TVar(s_2) = \emptyset;
• $s_2 \neq \text{"skip"}$;

We show that $S_1$ and $S'_2; s'_2; S''_2$ terminate in the same way when started in the states $m_1(f_1, \sigma_1)$, $m_2(f_2, \sigma_2)$ respectively by the hypothesis IH. The proof is same as that in case a).

We show that $s_2$ terminates if the execution of $S'_2; s'_2; S''_2$ terminates. We need to prove that $s'_2$ and $s_2$ start in the states agreeing on the values of variables in TVar(s_2). By assumption, $S'_2; s'_2; S''_2$ terminates, $(S'_2; s'_2; S''_2, m_2(f_2, \sigma_2)) \rightarrow (\text{skip}, m_2(f_2, \sigma'_2))$. Then, by Corollary B.3.1, $(S'_2; s'_2; S''_2, s_2, m_2(f_2, \sigma_2)) \rightarrow (s_2, m'_2(f_2, \sigma'_2))$. In addition, the execution of $S'_2$ and $s'_2$ must terminate because the execution of $S'_2; s'_2; S''_2$ terminates,

$(S'_2; s'_2; S''_2, s_2, m_2(f_2, \sigma_2)) \rightarrow (s'_2; S'_2; s_2, m_2(f_2, \sigma'_2)) \rightarrow (s'_2, m'_2(f_2, \sigma'_2))$.

By assumption, Def($s'_2; S''_2$) \cap TVar(s_2) = \emptyset. Then, by Corollary B.3.2, the value store $\sigma'_2$ and $\sigma'_2$ agree on values of the termination deciding variables of $s_2$, $\forall x \in TVar(s_2) : \sigma'_2(x) = \sigma'_2(x)$. By Corollary 4.4.1, TVar($s'_2$) = TVar($s_2$). Because the execution of $s'_2$ terminates, then the execution of $s_2$ terminates when started in the state $m_2'(f_2, \sigma'_2)$ by the hypothesis IH, $(s_2, m'_2(f_2, \sigma'_2)) \rightarrow (\text{skip}, m''_2)$.

In addition, we show that there is no input statement in $s_2$ by contradiction. Suppose there is input statement in $s_2$. By Lemma 4.4.8, $id_1 \in CVar(s_2)$. Hence, the input sequence variable is in the termination deciding variables of $s_2$, $id_1 \in TVar(s_2)$. By Corollary 4.4.1, TVar($s'_2$) = TVar($s_2$). Then, there must be one input statement in $s'_2$. Otherwise, by Lemma 4.4.2, the input sequence variable is not in the termination deciding variables of $s'_2$. A contradiction against the result that $id_1 \in TVar(s'_2)$.

Since there is one input statement in $s'_2$, by Lemma 4.4.8, $id_1 \in Def(s'_2)$. Thus, by definition, $id_1 \in Def(s'_2; S''_2)$. Then, Def($s'_2; S''_2$) \cap TVar($s_2$) \neq \emptyset. A contradiction. Therefore, there is no input statement in $s_2$.

(d) $S_1 = S'_1; s_1; s'_1$; and $S_2 = S'_2; s_2; s'_2$ where $s_1$ and $s_2$ are reordered and all of the following hold:
• $S'_1 \equiv^S_H S'_2$;
• $S'_1$ and $S'_2$ have equivalent computation of TVar($s_1; s'_1$) \cup TVar($s_2; s'_2$).
• $s_1 \equiv^S_H s'_2$;
• $s'_1 \equiv^S_H s_2$;
• Def($s_1$) \cap TVar($s'_1$) = \emptyset;
• Def($s_2$) \cap TVar($s'_2$) = \emptyset;

The proof is to show that if $S_1$ terminates when started in the state $m_1$, the $S_2$ terminates when started in the state $m_2$, and vice versa. Due to the symmetric conditions, it is suffice to show one direction that, w.l.o.g.,

$(S_1, m_1) \rightarrow (\text{skip}, m'_1) \Rightarrow (S_2, m_2) \rightarrow (\text{skip}, m'_2)$.
We show that the execution of \( S'_2 \) terminates by the hypothesis IH. We need to show that all required conditions are satisfied.

- \( \text{size}(S'_1) + \text{size}(S'_2) < k \).
  This is because \( \text{size}(s_1; s'_1) > 1, \text{size}(s_2; s'_2) > 1 \).

- Initial value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of the termination deciding variables of \( S'_1 \) and \( S'_2 \), \( \forall x \in \text{TVar}(S'_1) \cup \text{TVar}(S'_2) : \sigma_1(x) = \sigma_2(x) \).

We show that \( \text{TVar}(S'_1) \subseteq \text{TVar}(S_1) \). In the following, we prove that \( \text{CVar}(S'_1) \subseteq \text{CVar}(S_1) \).

\[
\text{CVar}(S'_1) \\
\subseteq \text{CVar}(S'_1; s_1) \text{ by the definition of } \text{CVar}(S'_1; s_1) \\
\subseteq \text{CVar}(S'_1; s_1; s'_1) \text{ by the definition of } \text{CVar}(S'_1; s_1; s'_1)
\]

Similarly, \( \text{LVar}(S'_1) \subseteq \text{LVar}(S_1) \). Hence, \( \text{TVar}(S'_1) \subseteq \text{TVar}(S_1) \). Similarly, \( \text{TVar}(S'_2) \subseteq \text{TVar}(S_2) \). By assumption, initial value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of the termination deciding variables of \( S'_1 \) and \( S'_2 \).

By the hypothesis IH, \( (S'_1, m_1(\sigma_1)) \equiv_H (S'_2, m_2(\sigma_2)) \). Because the execution of \( S_1 \) terminates, then \( S'_1 \) terminates when started in the state \( m_1(\sigma_1) \), \( (S'_1, m_1(\sigma_1)) \rightarrow (\text{skip}, m'_1(\sigma'_1)) \). Therefore, \( S'_2 \) terminates when started in the state \( m_2(\sigma_2) \), \( (S'_2, m_2(\sigma_2)) \rightarrow (\text{skip}, m'_2(\sigma'_2)) \).

We show that after the execution of \( S'_1 \) and \( S'_2 \), value stores agree on values of the termination deciding variables of \( s_1; s'_1 \) and \( s_2; s'_2 \), \( \forall x \in \text{TVar}(s_1; s'_1) \cup \text{TVar}(s_2; s'_2), \sigma'_1(x) = \sigma'_2(x). \) We split the argument into two steps.

i. We show that \( \text{TVar}(s_1; s'_1) = \text{TVar}(s_2; s'_2) \).
   By Corollary 4.4.1, \( \text{TVar}(s_1) = \text{TVar}(s'_2) \) and \( \text{TVar}(s'_1) = \text{TVar}(s_2) \).
   Then we show that \( \text{TVar}(s_1; s'_1) = \text{TVar}(s_1) \cup \text{TVar}(s'_1) \). To do that, we show that \( \text{CVar}(s_1; s'_1) = \text{CVar}(s_1) \cup \text{CVar}(s'_1) \).

\[
\text{CVar}(s_1; s'_1) \\
= \text{CVar}(s_1) \cup \text{Imp}(s_1, \text{CVar}(s'_1)) \text{ by the definition of } \text{CVar}(s_1; s'_1) \\
= \text{CVar}(s_1) \cup \text{CVar}(s'_1) \text{ by Def}(s_1) \cap \text{TVar}(s'_1) = \emptyset \text{ and } \text{the definition of } \text{Imp}(\cdot).
\]

Similarly, \( \text{LVar}(s_1; s'_1) = \text{LVar}(s_1) \cup \text{LVar}(s'_1) \). Thus, \( \text{TVar}(s_1; s'_1) = \text{TVar}(s_1) \cup \text{TVar}(s'_1) \). Similarly, \( \text{TVar}(s_2; s'_2) = \text{TVar}(s_2) \cup \text{TVar}(s'_2) \).

In summary, \( \text{TVar}(s_1; s'_1) = \text{TVar}(s_2; s'_2) \).

ii. We show that \( \text{Imp}(S'_1, \text{TVar}(s_1; s'_1)) \subseteq \text{TVar}(S_1) \) and \( \text{Imp}(S'_2, \text{TVar}(s_2; s'_2)) \subseteq \text{TVar}(S_2) \).

W.l.o.g., we show that \( \text{Imp}(S'_1, \text{TVar}(s_1; s'_1)) \subseteq \text{TVar}(S_1) \).

Specifically, we show \( \text{Imp}(S'_1, \text{CVar}(s_1; s'_1)) \subseteq \text{CVar}(S_1) \).

\[
\text{Imp}(S'_1, \text{CVar}(s_1; s'_1)) \\
= \text{Imp}(S'_1, \text{CVar}(s_1) \cup \text{Imp}(s_1, \text{CVar}(s'_1))) \text{ by (1)} \\
= \text{Imp}(S'_1, \text{CVar}(s_1)) \cup \text{Imp}(S'_1, \text{Imp}(s_1, \text{CVar}(s'_1))) \text{ by (2)}
\]

197
by Lemma B.1.2

$$\text{Imp}(S'_1, \text{CVar}(s_1))$$
$$\subseteq \text{CVar}(S'_1; s_1)$$ by the definition of CVar(·)
$$\subseteq \text{CVar}(S'_1; s_1; s'_1)$$ by the definition of CVar(·)

$$\text{Imp}(S'_1, \text{Imp}(s_1, \text{CVar}(s'_1)))$$
$$= \text{Imp}(S'_1; s_1, \text{CVar}(s'_1))$$ by Lemma B.1.1
$$\subseteq \text{CVar}(S'_1; s_1; s'_1)$$ by the definition of CVar(·).

$$\text{Imp}(S'_1, \text{CVar}(s_1)) \cup \text{Imp}(S'_1, \text{Imp}(s_1, \text{CVar}(s'_1)))$$
$$\subseteq \text{CVar}(S'_1; s_1; s'_1)$$ by (3) and (4).

In conclusion, Imp($S'_1, \text{CVar}(s_1; s'_1)$) ⊆ CVar($S_1$). Similarly, Imp($S'_1, \text{LVar}(s_1; s'_1)$) ⊆ LVar($S_1$). Thus, Imp($S'_1, \text{TVar}(s_1; s'_1)$) ⊆ TVar($S_1$). Similarly, Imp($S'_2, \text{TVar}(s_2; s'_2)$) ⊆ TVar($S_2$).

Then, by Theorem 2, after terminating execution of $S'_1$ and $S'_2$, value stores $\sigma'_1$ and $\sigma'_2$ agree on values of the termination deciding variables of $s_1; s'_1$ and $s_2; s'_2$, $\forall x \in \text{TVar}(s_1; s'_1) \cup \text{TVar}(s_2; s'_2) : \sigma'_1(x) = \sigma'_2(x)$.

We show that the execution of $s_2$ terminates by the hypothesis IH. By Corollary B.3.1, ($S'_1; s_1; s'_1, m_1(\sigma_1)$) $\rightarrow (s_1; s'_1, m'_1(\sigma'_1))$ and ($S'_2; s_2; s'_2, m_2(\sigma_2)$) $\rightarrow (s_2; s'_2, m'_2(\sigma'_2))$. By assumption, $S_1$ terminates, then $s_1$ terminates, $(s_1, m'_1(\sigma'_1))$ $\rightarrow$ (skip, $m'_1(\sigma''_1)$). Because $s'_1 \equiv_H s_2$, to apply the induction hypothesis, we need to show that all required conditions hold.

- $\text{size}(s_2) + \text{size}(s'_1) < k$.
  By definition, $\text{size}(S'_2) > 1, \text{size}(S'_1) > 1, \text{size}(s_1) > 1, \text{size}(s'_2) > 1$.
- Value stores $\sigma''_1$ and $\sigma'_2$ agree on values of the termination deciding variables of $s'_1$ and $s_2$. $\forall x \in \text{TVar}(s'_1) \cup \text{TVar}(s_2) : \sigma''_1(x) = \sigma'_2(x)$.

By Corollary 4.4.1, TVar($s'_1$) = TVar($s_2$). Because of the condition Def($s_1) \cap \text{TVar}(s'_1) = \emptyset$, by Corollary B.3.2, value stores $\sigma''_1$ and $\sigma'_2$ agree on values of the termination deciding variables of $s'_1$. $\forall x \in \text{TVar}(s'_1) : \sigma''_1(x) = \sigma'_1(x)$. By the argument above, $\forall x \in \text{TVar}(s_2) : \sigma'_1(x) = \sigma'_2(x)$. Thus, the condition holds.

By the induction hypothesis IH, ($s'_1, m''_1(\sigma''_1)$) $\equiv_H (s_2, m'_2(\sigma'_2))$. Because the execution of $s'_1$ terminates, then the execution of $s_2$ terminates when started in the state $m'_2(\sigma'_2)$. ($s_2, m'_2(\sigma'_2)$) $\rightarrow$ (skip, $m''_2(\sigma''_2)$).

We show that the execution of $s'_2$ terminates. This is by the similar argument that $s_2$ terminates.

In conclusion, $S_2$ terminates when started in the state $m_2(\sigma_2)$. The theorem holds.

In addition, we show that it is impossible that $s_1$ and $s'_1$ both include input statements by contradiction. Suppose there are input statements in both $s_1$ and $s'_1$. By Lemma 4.4.8, $id_1 \in \text{Def}(s_1) \cap \text{TVar}(s'_1)$. A contradiction against the condition that Def($s_1) \cap \text{TVar}(s'_1) = \emptyset$. Similarly, there are no input statements in both $s_2$ and $s'_2$.
APPENDIX E

THE PROOF FOR THE SAME I/O THEOREM
Theorem 5: Two statement sequences \( S_1 \) and \( S_2 \) satisfy the proof rule of the behavioral equivalence, \( S_1 \equiv^S \ O \ S_2 \). If \( S_1 \) and \( S_2 \) start in states \( m_1(f_1, \sigma_1) \) and \( m_2(f_2, \sigma_2) \) where both of the following hold:

- Crash flags are not set, \( f_1 = f_2 = 0 \);
- Value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of the output deciding variables of \( S_1 \) and \( S_2 \), \( \forall id \in \text{OVar}(S_1) \cup \text{OVar}(S_2) : \sigma_1(id) = \sigma_2(id) \);

then \( S_1 \) and \( S_2 \) produce the same output sequence, \( (S_1, m_1) \equiv_O (S_2, m_2) \).

Proof.

The proof is by induction on the sum of program size of \( S_1 \) and \( S_2 \), \( \text{size}(S_1) + \text{size}(S_2) \) and is a case analysis based on \( S_1 \equiv^S \ O \ S_2 \).

Base case.

\( S_1 \) and \( S_2 \) are simple statements. There are two cases according to the proof rule of behavioral equivalence because stacks are not changed in executions of \( S_1 \) and \( S_2 \).

1. \( S_1 \) and \( S_2 \) are not output statement, \( \forall e_1 e_2 : (\text{"output } e_1 \text{"} \neq S_1) \land (\text{"output } e_2 \text{"} \neq S_2) \);

By the definition of imported variables relative to output, \( \text{Imp}_o(S_1) = \text{Imp}_o(S_2) = \{id_{IO}\} \). By assumption, initial value stores \( \sigma_1 \) and \( \sigma_2 \) agree on the value of the I/O sequence variable, \( \sigma_1(id_{IO}) = \sigma_2(id_{IO}) \). By definition, \( \text{Out}(\sigma_1) = \text{Out}(\sigma_2) \).

By Lemma 4.5.7, in any state \( m_1' \) reachable from \( m_1 \), the output sequence in \( m_1' \) is same as that in \( m_1 \), \( \forall m_1' : ((S_1, m_1(\sigma_1)) \xrightarrow{*} (S_1', m_1'(\sigma_1'))) \Rightarrow (\text{Out}(\sigma_1') = \text{Out}(\sigma_1)) \). Similarly, for any state \( m_2' \) reachable from \( m_2 \), the output sequence in \( m_2' \) is same as that in \( m_2 \). The theorem holds.

2. \( S_1 = S_2 = \text{"output } e \" \).

We show that the expression \( e \) evaluates to the same value w.r.t value stores, \( \sigma_1, \sigma_2 \). By the definition of imported variables relative to output, \( \text{Imp}_o(S_1) = \text{Imp}_o(S_2) = \text{Use}(e) \cup \{id_{IO}\} \). Then \( \forall x \in \text{Use}(e) \cup \{id_{IO}\} : \sigma_1(x) = \sigma_2(x) \) by assumption. By Lemma B.2.1, \( \mathcal{E}[e]_{\sigma_1} = \mathcal{E}[e]_{\sigma_2} \). Then, there are two possibilities.

(a) \( \mathcal{E}[e]_{\sigma_1} = \mathcal{E}[e]_{\sigma_2} = (\text{error}, v_{error}) \).

The execution of \( S_1 \) proceeds as follows.

\[
\begin{align*}
\text{(output } e, m_1(\sigma_1)) \\
= \text{(output } (\text{error}, v_{error}), m_1(\sigma_1)) \text{ by the rule EEval'} \\
\rightarrow \text{(output } 0, m_1(1/f)) \text{ by the ECrash rule} \\
\rightarrow \text{(output } 0, m_1(1/f)) \text{ for any } i > 0 \text{ by the Crash rule}.
\end{align*}
\]

Similarly, the execution of \( S_2 \) does not terminate and there is no change to I/O sequence in execution. Because \( \sigma_1(id_{IO}) = \sigma_2(id_{IO}) \), then the output sequence in value stores \( \sigma_1 \) and \( \sigma_2 \) are same, \( \text{Out}(\sigma_1) = \text{Out}(\sigma_2) \), the theorem holds.
(b) \( E[e]_{\sigma_1} = E[e]_{\sigma_2} \neq (\text{error}, v_{ef}) \)

(1) \( S_1 \) and \( S_2 \) satisfy the proof rule of equivalent computation of I/O sequence variable and their initial states agree on the values of the imported variables relative to I/O sequence variable. By Theorem 2, \( S_1 \) and \( S_2 \) produce the same output sequence after terminating execution when started in state \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) respectively. The theorem holds.

**Induction step.**

The hypothesis IH is that Theorem 5 holds when \( \text{size}(S_1) + \text{size}(S_2) = k \geq 2 \).

We show Theorem 5 holds when \( \text{size}(S_1) + \text{size}(S_2) = k + 1 \). The proof is a case analysis according to the cases in the definition of the proof rule of behavioral equivalence.

1. \( S_1 \) and \( S_2 \) are one statement and one of the following holds:

   (a) \( S_1 = \text{"\text{If}(e) then } \{S^f_1\} \text{ else } \{S^t_1\}\" \) and \( S_2 = \text{"\text{If}(e) then } \{S^f_2\} \text{ else } \{S^t_2\}\" \)

   and all of the following hold:

   - There is an output statement in \( S_1 \) and \( S_2 \): \( \exists e_1 e_2 : (\text{"output } e_1 \" \in S_1) \land (\text{"output } e_2 \" \in S_2) \);
   - \( S^t_1 \equiv^S_0 S^t_2 \);
   - \( S^f_1 \equiv^S_0 S^f_2 \);

   By Lemma 4.5.1, \( \{id_{1O}\} \in \text{Imp}_o(S_1) \). By assumption, value stores \( \sigma_1 \) and \( \sigma_2 \) agree on the value of the I/O sequence variable and the I/O sequence variable, \( \sigma_1(id_{1O}) = \sigma_2(id_{1O}) \).

   We show that the evaluations of the predicate expression of \( S_1 \) and \( S_2 \) w.r.t. initial value store \( \sigma_1 \) and \( \sigma_2 \) produce the same value. We need to show that value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of variables used in the predicate expression \( e \) of \( S_1 \) and \( S_2 \). Because the output sequence is defined in \( S_1 \), by the definition of imported variables relative to output, \( \text{Imp}_o(S_1) = \text{Use}(e) \cup \text{Imp}_o(S^t_1) \cup \text{Imp}_o(S^f_1) \). Thus, \( \text{Use}(e) \subseteq \text{OVar}(S_1) \). By assumption, value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of variables used in the predicate expression \( e \) of \( S_1 \) and \( S_2 \), \( \forall x \in \text{Use}(e) : \sigma_1(x) = \sigma_2(x) \). By Lemma B.2.1, the evaluations of the predicate expression of \( S_1 \) and \( S_2 \) w.r.t. pairs value stores, \( \sigma_1 \) and \( \sigma_2 \) generate the same value, \( E'[e]_{\sigma_1} = E'[e]_{\sigma_2} \).

   Then there are two possibilities.

   i. \( E'[e]_{\sigma_1} = E'[e]_{\sigma_2} = (\text{error}, v_{ef}) \).

   Then the execution of \( S_1 \) proceeds as follows:

   \[
   (\text{If}(e) \text{ then } \{S^t_1\} \text{ else } \{S^f_1\}, m_1(\sigma_1))
   \]

   \[
   \rightarrow (\text{If}(\text{error}, v_{ef}) \text{ then } \{S^t_1\} \text{ else } \{S^f_1\}, m_1(\sigma_1))
   \]

   by the EEval' rule

   \[
   \rightarrow (\text{If}(0) \text{ then } \{S^t_1\} \text{ else } \{S^f_1\}, m_1(1/f))
   \]

   by the ECrash rule

   \[
   \rightarrow^i (\text{If}(0) \text{ then } \{S^t_1\} \text{ else } \{S^f_1\}, m_1(1/f))
   \]

   for any \( i > 0 \), by the Crash rule.
Similarly, the execution of $S_2$ does not terminate and does not redefine I/O sequence. Because $\sigma_1(id_{IO}) = \sigma_2(id_{IO})$, the theorem holds.

ii. $E'[e]\sigma_1 = E'[e]\sigma_2 \neq (\text{error}, v_{ef})$. W.l.o.g., $E'[e]\sigma_1 = E'[e]\sigma_2 = (0, v_{ef})$. The execution of $S_1$ proceeds as follows.

$$
\begin{align*}
(\text{If}(e) \text{ then } \{S_1^f\} \text{ else } \{S_2^f\}, m_1(\sigma_1)) \\
\rightarrow (\text{If}(0, v_{ef}) \text{ then } \{S_1^f\} \text{ else } \{S_2^f\}, m_1(\sigma_1)) \\
\quad \text{by the EEval rule} \\
\rightarrow (\text{If}(0) \text{ then } \{S_1^f\} \text{ else } \{S_2^f\}, m_1(\sigma_1)) \\
\quad \text{by the E-Else1 or E-Else2 rule} \\
\rightarrow (S_1^f, m_1(\sigma_1)) \text{ by the If-F rule}.
\end{align*}
$$

Similarly, the execution of $S_2$ proceeds to $(S_2^f, m_2(\sigma_2))$ after two steps. By the hypothesis IH, we show that $S_1^f$ and $S_2^f$ produce the same output sequence when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$. We need to show that all required conditions are satisfied.

- $\text{size}(S_1^f) + \text{size}(S_2^f) \leq k$.
  By definition, $\text{size}(S_1) = 1 + \text{size}(S_1^f) + \text{size}(S_2^f)$. Therefore, $\text{size}(S_1^f) + \text{size}(S_2^f) < k$.

- Value stores $\sigma_1$ and $\sigma_2$ agree on values of the out-deciding variables of $S_1^f$ and $S_2^f$, $\forall x \in \text{OVar}(S_1^f) \cup \text{OVar}(S_2^f) : \sigma_1(x) = \sigma_2(x)$.

  By the definition of imported variables relative to output, $\text{Imp}_o(S_1^f) \subseteq \text{Imp}_o(S_1)$. Besides, by the definition of TVar $\rho_o(S_1)$, $\text{TVar}_o(S_1^f) \subseteq \text{TVar}_o(S_1)$. Then $\text{OVar}(S_1^f) \subseteq \text{OVar}(S_1)$. Similarly, $\text{OVar}(S_2^f) \subseteq \text{OVar}(S_2)$. By assumption, the value stores $\sigma_1$ and $\sigma_2$ agree on the values of the out-deciding variables of $S_1^f$ and $S_2^f$.

By the hypothesis IH, $S_1^f$ and $S_2^f$ produce the same output sequence when started from state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. The theorem holds.

(b) $S_1 = \text{"while}_{(n_1)}(e) \{S_1''\}$ and $S_2 = \text{"while}_{(n_2)}(e) \{S_2''\}$ and all of the following hold:

- There is an output statement in $S_1$ and $S_2$: $\exists e_1 e_2 : (\text{"output } e_1\text{" }\in S_1) \land (\text{"output } e_2\text{" }\in S_2)$;
- $S_1'' = S_2''$;
- Both loop bodies satisfy the proof rule of termination in the same way: $S_1'' = S_2''$;
- $S_1''$ and $S_2''$ have equivalent computation of $\text{OVar}(S_1) \cup \text{OVar}(S_2)$;

By Corollary 4.5.2, we show that $S_1$ and $S_2$ produce the same output sequence when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. We need to show that the required conditions are satisfied.

- Crash flags are not set, $f_1 = f_2 = 0$.
- Value stores $\sigma_1$ and $\sigma_2$ agree on the values of the out-deciding variables of $S_1$ and $S_2$, $\forall x \in \text{OVar}(S_1) \cup \text{OVar}(S_2) : \sigma_1(x) = \sigma_2(x)$.
2. The loop counter value of \( S_1 \) and \( S_2 \) are zero in initial loop counter, 
\[ \text{loop}_c^1(n_1) = \text{loop}_c^2(n_2) = 0. \]
2. The loop body of \( S_1 \) and \( S_2 \) satisfy the proof rule of termination in the same way, \( S''_1 \equiv_S S''_2. \)
2. The loop body of \( S_1 \) and \( S_2 \) satisfy the proof rule of equivalent computation of \( \text{OVar}(S_1) \cup \text{OVar}(S_2). \)
2. The above five conditions are from assumption.
2. \( S_1 \) and \( S_2 \) have same set of termination deciding variables, \( \text{TVar}(S_1) = \text{TVar}(S_2). \)
By the definition of \( \text{TVar}_o(S_1) \), \( \text{TVar}_o(S_1) = \text{TVar}(S_1) \) and \( \text{TVar}_o(S_2) = \text{TVar}(S_2). \) By Lemma 4.5.5, \( \text{TVar}_o(S_1) = \text{TVar}_o(S_2). \) Thus, \( \text{TVar}(S_1) = \text{TVar}(S_2). \)
2. \( S_1 \) and \( S_2 \) have same set of imported variables relative to the I/O sequence variable, 
\[ \text{Imp}(S_1, \{id_{10}\}) = \text{Imp}(S_2, \{id_{10}\}). \]
By Lemma 4.5.3, \( \text{Imp}_o(S_1) = \text{Imp}_o(S_2). \) By definition, \( \text{Imp}_o(S_1) = \text{Imp}_o(S_1, \{id_{10}\}) \) and \( \text{Imp}_o(S_2) = \text{Imp}(S_2, \{id_{10}\}). \) Thus, 
\[ \text{Imp}(S_1, \{id_{10}\}) = \text{Imp}(S_2, \{id_{10}\}). \]
2. The loop body of \( S_1 \) and \( S_2 \) produce the same output sequence when started in states with crash flags not set and whose value stores agree on values of the out-deciding variables of \( S''_1 \) and \( S''_2 \), \( \forall m_{S''_1}(f''_1, \sigma''_1) m_{S''_2}(f''_2, \sigma''_2) : (\forall x \in \text{OVar}(S''_1) \cup \text{OVar}(S''_2) : \sigma''_1(x) = \sigma''_2(x)) \land (f''_1 = f''_2 = 0) \Rightarrow (S''_1, m_{S''_1}(f''_1, \sigma''_1)) \equiv o (S''_2, m_{S''_2}(f''_2, \sigma''_2)). \)
Because size\( (S_1) = 1 + \text{size}(S''_1), \text{size}(S_2) = 1 + \text{size}(S''_2), \) then size\( (S''_1) + \text{size}(S''_2) < k. \) By the hypothesis IH, the condition is satisfied.
By Corollary 4.5.2, \( S_1 \) and \( S_2 \) produce the same output sequence when started in states \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) respectively. The theorem holds.
(c) Output statements are not in both \( S_1 \) and \( S_2 \), \( \forall e_1 e_2 : (\text{“output } e_1 \text{”} \notin S_1) \land (\text{“output } e_2 \text{”} \notin S_2) \).
By Lemma 4.5.1, \( \{id_{10}\} \subseteq \text{Imp}_o(S_1) \). By assumption, value stores in initial states \( m_1, m_2 \) agree on values of the I/O sequence variable, \( \gamma(\{id_{10}\}) = \sigma_2(\text{id}_{10}) \). By Lemma 4.5.7, the value of output sequence is same in \( m_1 \) and any state reachable from \( m_1 \), \( \forall m'_1 m'_2 \ : (S_1, m_1(\sigma_1)) \equiv o (S'_1, \text{Out}(\sigma_1)) \) and \( (S_2, m_2(\sigma_2)) \equiv o (S'_2, \text{Out}(\sigma_2)), \text{Out}(\sigma_1) \) = \text{Out}(\sigma_1) = \text{Out}(\sigma_2) = \text{Out}(\sigma_2) \). The theorem holds.

2. \( S_1 = S'_1; s_1 \) and \( S_2 = S'_2; s_2 \) are not both one statement and one of the following holds:

(a) There is an output statement in both \( s_1 \) and \( s_2 \), \( \exists e_1 e_2 : (\text{“output } e_1 \text{”} \in s_1) \land (\text{“output } e_2 \text{”} \in s_2), \) and all of the following hold:

- \( S'_1 \equiv_S S'_2; \)
- \( S'_1 \) and \( S'_2 \) satisfy the proof rule of termination in the same way: \( S'_1 \equiv_S S'_2; \)
• \( S'_1 \) and \( S'_2 \) have equivalent computation of \( \text{OVar}(s_1) \cup \text{OVar}(s_2) \);
• \( s_1 \equiv_O^S s_2 \);

By the hypothesis IH, we show \( S'_1 \) and \( S'_2 \) produce the same output sequence when started in states \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) respectively. We need to show that all required conditions are satisfied.

• \( \text{size}(S'_1) + \text{size}(S'_2) < k \).
  By the definition of program size, \( \text{size}(s_1) \geq 1, \text{size}(s_2) \geq 1 \). Then \( \text{size}(S'_1) + \text{size}(S'_2) < k \).

• Value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of the out-deciding variables of \( S'_1 \) and \( S'_2 \), \( \forall x \in \text{OVar}(S'_1) \cup \text{OVar}(S'_2) : \sigma_1(x) = \sigma_2(x) \).

We show that \( \text{TVar}_o(S'_1) \subseteq \text{TVar}_o(S_1) \).

\[
\text{TVar}_o(S'_1) \\
\subseteq \text{TVar}(S'_1) \quad \text{by Lemma 4.5.4} \\
\subseteq \text{TVar}_o(S_1) \quad \text{by the definition of } \text{TVar}_o(S_1) 
\]

We show that \( \text{Imp}_o(S'_1) \subseteq \text{Imp}_o(S_1) \).

\[
\text{Imp}_o(S'_1) \\
\subseteq \text{Imp}(S'_1; \{id_{IO}\}) \quad (1) \quad \text{by Lemma 4.5.2} \\
\{id_{IO}\} \subseteq \text{Imp}_o(s_{k+1}) \quad (2) \quad \text{by Lemma 4.5.1} \\
\]

Combining (1) + (2)

\[
\text{Imp}(S'_1; \{id_{IO}\}) \\
\subseteq \text{Imp}(S'_1, \text{Imp}_o(s_1)) \quad \text{by Lemma B.1.2} \\
= \text{Imp}_o(S_1) \quad \text{by the definition of } \text{Imp}_o(S). 
\]

Similarly, \( \text{OVar}(S'_2) \subseteq \text{OVar}(S_2) \). By assumption, value stores \( \sigma_1 \) and \( \sigma_2 \) agree on values of out-deciding variables of \( S'_1 \) and \( S'_2 \).

By the hypothesis IH, \( S'_1 \) and \( S'_2 \) produce the same output sequence when started in state \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) respectively.

We show that \( S_1 \) and \( S_2 \) produce the same output sequence if \( s_1 \) and \( s_2 \) execute. We need to show that \( S'_1 \) and \( S'_2 \) terminate in the same way when started in states \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) respectively. Specifically, we prove that the value stores \( \sigma_1 \) and \( \sigma_2 \) agree on the values of termination deciding variables of \( S'_1 \) and \( S'_2 \). By definition, the termination deciding variables in \( S'_t \) are a subset of the termination deciding variables relative to output, \( \text{TVar}(S'_1) \subseteq \text{TVar}_o(S_1) \). Similarly, \( \text{TVar}(S'_2) \subseteq \text{TVar}_o(S_2) \).

By assumption, the value stores \( \sigma_1 \) and \( \sigma_2 \) agree on the values of the termination deciding variables of \( S'_1 \) and \( S'_2 \), \( \forall x \in \text{TVar}(S'_1) \cup \text{TVar}(S'_2) : \sigma_1(x) = \sigma_2(x) \). By Theorem 4, \( S'_1 \) and \( S'_2 \) terminate in the same way when started in state \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \) respectively.

If \( S'_1 \) and \( S'_2 \) terminate when started in states \( m_1(\sigma_1) \) and \( m_2(\sigma_2) \), by Lemma 4.4.11, \( S'_1 \) and \( S'_2 \) consume same amount of input values. In addition, we show that value stores agree on values of the out-deciding variables of \( s_1 \) and \( s_2 \) by Theorem 2. We need to show that \( S'_1 \) and \( S'_2 \) start execution in states agreeing on values of the imported variables in \( S'_{1} \) and \( S'_{2} \) relative to the out-deciding variables of \( s_1 \) and \( s_2 \).
• $\text{Imp}(\text{TVar}_o(s_1), S'_1) \subseteq \text{TVar}_o(S_1)$.
  This is by the definition of $\text{TVar}_o(S_1)$.

• $\text{Imp}(\text{Imp}_o(s_1), S'_1) = \text{Imp}_o(S_1)$.
  This is by the definition of $\text{Imp}_o(S_1)$.

Thus, the imported variables in $S'_1$ relative to the out-deciding variables of $s_1$ are a subset of the out-deciding variables of $S_1$, $\text{Imp}(S'_1, \text{OVar}(s_1)) \subseteq \text{OVar}(S_1)$. Similarly, $\text{Imp}(S'_2, \text{OVar}(s_2)) \subseteq \text{OVar}(S_2)$. By Corollary 4.5.1, $s_1$ and $s_2$ have same out-deciding variables, $\text{OVar}(s_1) = \text{OVar}(s_2)$. By assumption, $S'_1$ and $S'_2$ terminate when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$, $(S'_1, m_1(\sigma_1)) \xrightarrow{\text{step}} (\text{skip}, m'_1(\sigma'_1)), (S'_2, m_1(\sigma_2)) \xrightarrow{\text{step}} (\text{skip}, m'_2(\sigma'_2))$. By Theorem 2, value stores $\sigma'_1$ and $\sigma'_2$ agree on values of the out-deciding variables of $s_1$ and $s_2$.

By the hypothesis IH again, $s_1$ and $s_2$ produce the same output sequence when started in states $m'_1(\sigma'_1)$ and $m'_2(\sigma'_2)$ respectively. The theorem holds.

(b) There is no output statement in the last statement in $S_1$ or $S_2$: W.l.o.g., $(\forall e : \text{“output e" } \notin s_1) \land ((S'_1) = (S_2)).$

By the hypothesis IH, we show that $S'_1$ and $S_2$ produce the same output sequence when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. We need to show that two required conditions are satisfied.

• $\text{size}(S'_1) + \text{size}(S_2) \leq k$.
  $\text{size}(s_1) \geq 1$ by definition. Then $\text{size}(S'_1) + \text{size}(S_2) \leq k$.

• $\forall x \in \text{OVar}(S'_1) \cup \text{OVar}(S_2) : \sigma_1(x) = \sigma_2(x)$.
  By definition of $\text{TVar}_o(S)/\text{Imp}_o(S)$, $\text{TVar}_o(S'_1) = \text{TVar}_o(S_1)$, and $\text{Imp}_o(S'_1) = \text{Imp}_o(S_1)$ Hence, $\forall x \in \text{OVar}(S'_1) \cup \text{OVar}(S_2) : \sigma_1(x) = \sigma_2(x)$.

Therefore, $S'_1$ and $S_2$ produce the same output sequence when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively, $(S'_1, m_1) \equiv_O (S_2, m_2)$ by the hypothesis IH.

When the execution of $S'_1$ terminates, then the output sequence is not changed in the execution of $s_1$ by Lemma 4.5.7. The theorem holds.