HAL
open science

Certified algorithms for program slicing

Jean-Christophe Léchenet

To cite this version:

Jean-Christophe Léchenet. Certified algorithms for program slicing. Autre. Université Paris-Saclay, 2018. Français. NNT : 2018SACLC056 . tel-01874620v1

HAL Id: tel-01874620

https://theses.hal.science/tel-01874620v1

Submitted on 14 Sep 2018 (v1), last revised 29 Nov 2018 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

université PARIS-SACLAY

CentraleSupélec

NNT : 2018SACLC056

Certified Algorithms for Program Slicing

Thèse de doctorat de l'Université Paris-Saclay
préparée à CentraleSupélec

École doctorale n° 573 Interfaces
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 19 juillet 2018, par

Jean-Christophe Léchenet

Composition du jury :

Claude Marché Directeur de recherche, Inria & Université Paris-Saclay	Président
Torben Amtoft Professeur, Kansas State University	Rapporteur
David Pichardie Professeur, ENS Rennes	Rapporteur
Catherine Dubois Professeur, ENSIIE	Examinatrice
Frédéric Loulergue Professeur, Northern Arizona University	Examinateur
Pascale Le Gall Professeur, CentraleSupélec	Directrice de thèse
Nikolai Kosmatov Ingénieur-chercheur, CEA List	Co-encadrant

Thèse de doctorat

Certified Algorithms for Program Slicing

Thèse de doctorat de l'Université Paris-Saclay
préparée à CentraleSupélec et au CEA List

École doctorale n° 573 Interfaces
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 19 juillet 2018, par

Jean-Christophe Léchenet

Composition du jury :

Claude Marché Directeur de recherche, Inria & Université Paris-Saclay	Président
Torben Amtoft Professeur, Kansas State University	Rapporteur
David Pichardie Professeur, ENS Rennes	Rapporteur
Catherine Dubois Professeur, ENSIIE	Examinatrice
Frédéric Loulergue Professeur, Northern Arizona University	Examinateur
Pascale Le Gall Professeur, CentraleSupélec	Directrice de thèse
Nikolai Kosmatov Ingénieur-chercheur, CEA List	Co-encadrant

Remerciements

Je remercie Torben Amtoft et David Pichardie d'avoir accepté de rapporter ma thèse. Je remercie Torben également d'avoir accepté de participer au jury bien qu'étant en vacances au Kenya avec une connexion instable, ce qui n'a pas dû être très facile. Thank you Torben! Je remercie également les examinateurs, Catherine Dubois et Frédéric Loulergue, et le président du jury, Claude Marché. Je remercie l'ensemble du jury de m'avoir permis de corriger un certain nombre d'imprécisions et d'erreurs contenues dans le manuscrit. Je remercie bien entendu mes deux encadrants : mon encadrant au CEA, Nikolai Kosmatov, qui m'a accompagné au jour le jour pendant ces trois ans et demi de thèse, et ma directrice de thèse, Pascale Le Gall, qui, bien que suivant de manière plus éloignée mes travaux, a su apporter son regard critique et son expérience du monde de la recherche.

Je remercie les personnes qui m'ont conduite à faire cette thèse. Tout d'abord, Marc Aiguier, qui lors d'un de ses cours à Centrale a conseillé à la classe de faire des thèses et m'a mis en contact avec Pascale. Et, bien entendu, Pascale, qui m'a encadré pour mon stage de fin d'études avant de me mettre en contact avec le CEA pour ma thèse.

Je remercie également toutes les personnes qui ont permis de créer un cadre agréable au CEA. En premier lieu, mes voisins de bureau au LSL : Vincent, qui était là dès le début mais seulement à mi-temps, puis Lionel qui nous a rejoint à plein temps. Je remercie plus largement tous les doctorants du LSL : David, expert en films et en jeux vidéos indépendants, qui, tel Gandalf, est passé de David le doctorant à David le permanent ; Allan, expert en C++ ; Steven, qui m'a fait refaire un peu d'algèbre ; Hugo, le roi des bons plans ; Benjamin, et ses remarques acides ; Quentin, pour ses prétendues chouquettes ; Alexandre, pour nos discussions sur l'e-sport ; Yaëlle et Maxime qui ont commencé plus récemment. Je remercie également les permanents, et en particulier André, qui a su animer les pauses café avec ses théories les plus farfelues. Je remercie aussi les quelques personnes du LISE que je connaissais : Gabriel, Imen, Nassim et Ngo Minh Thang.

Bien que n'y venant que très rarement, j'avais toujours un pied à Centrale (puis CentraleSupélec) par ma directrice de thèse. Je remercie les quelques personnes que je croisais lors de mes venues et avec lesquelles j'étais toujours heureux d'échanger

quelques mots : Hichame, Gauthier, Marc, Laurent et Céline.

Je remercie également mes amis de Centrale (alors Centrale Paris et non CentraleSupélec) : François, qui malheureusement s'est perdu de l'autre côté de l'Atlantique mais avec qui j'ai le plaisir de discuter régulièrement, et Maxime pour nos inoubliables soirées pizza/blind test, qui vont grandement me manquer. Je remercie également la bande de Sainte-Geneviève avec qui je partage des restaurants régulièrement : Bruno, Matthieu, Simon, Maxence, Hélène et Nikolas.

Enfin, je remercie ma famille, en premier lieu pour avoir continué à m'héberger pendant ma thèse, et pour leurs conseils avisés, comme : « ce n'est pas grave, tu as déjà publié » et, à propos d'un choix important, « tu n'as qu'à tirer à pile ou face ». Merci, enfin, à Agnès et Jean-Gabriel de s'être déplacés pour ma soutenance.

Résumé étendu en français

La simplification syntaxique, ou slicing, est une technique permettant d'extraire, à partir d'un programme et d'un critère consistant en une ou plusieurs instructions de ce programme, un programme plus simple, appelé slice, ayant le même comportement que le programme initial vis-à-vis de ce critère. Dans cette thèse, nous nous intéressons à la forme initiale du slicing introduite par Mark Weiser en 1981, appelée slicing statique arrière.

Les méthodes d'analyse de code permettent d'établir les propriétés d'un programme. Ces méthodes sont souvent coûteuses, et leur complexité augmente rapidement avec la taille du code. Il serait donc souhaitable d'appliquer ces techniques sur des slices plutôt que sur le programme initial, mais cela nécessite de pouvoir justifier théoriquement l'interprétation des résultats obtenus sur les slices.

Cette thèse apporte cette justification pour le cas de la recherche d'erreurs à l'exécution. Dans ce cadre, deux questions se posent. Si une erreur est détectée dans une slice, cela veut-il dire qu'elle se déclenchera aussi dans le programme initial ? Et inversement, si l'absence d'erreurs est prouvée dans une slice, cela veut-il dire que le programme initial en est lui aussi exempt ?

Dans un premier temps, nous rappelons les différents concepts du slicing statique arrière classique sur un mini-langage impératif. Classiquement, le calcul de la slice s'appuie sur deux relations de dépendance : les dépendances de contrôle et les dépendances de données. Les dépendances de contrôle relient une instruction aux instructions pouvant décider de son exécution (par exemple, dans notre langage, les instructions dans les branches d'une condition ont une dépendance de contrôle envers la condition). Les dépendances de données relient une instruction aux affectations pouvant modifier la valeur d'une de ses variables. En se basant sur ses deux dépendances, on peut calculer l'ensemble des instructions à préserver dans la slice, aussi appelé slice set : c'est l'ensemble des instructions dont dépend directement ou indirectement le critère de slicing. En partant du programme initial et du slice set, on peut construire la slice en supprimant les instructions qui ne sont pas contenues dans le slice set. Comme exprimé ci-dessus, la slice est censée avoir le même comportement que le programme initial vis-à-vis du critère de slicing. Ce lien est formalisé par la propriété de correction qui relie la sémantique du pro-

gramme initial et celle de la slice. Informellement, si le programme initial termine sur une entrée donnée, alors sa slice termine aussi sur cette entrée et les exécutions du programme et de sa slice s'accordent après chaque instruction préservée dans la slice sur les valeurs des variables apparaissant dans cette instruction. Formellement, cette propriété de correction s'écrie comme une égalité des projections des trajectoires du programme initial et de sa slice.

Pour répondre aux deux questions listées plus haut, nous étendons ce mini-langage de façon à ce qu'il soit représentatif pour notre problème. Pour ce faire, nous devons introduire des erreurs dans le langage et établir une propriété de correction qui décrit aussi les exécutions infinies. Nous introduisons les erreurs sous la forme d'assertions qui stoppent l'exécution du programme lorsque leur condition n'est pas vérifiée et n'ont aucun effet sinon. Nous faisons l'hypothèse que seules les assertions peuvent produire des erreurs, les autres instructions devant être protégées par des assertions si elles contiennent des expressions dites menaçantes, c'est-à-dire dont l'évaluation peut conduire à une erreur. Nous réutilisons le cadre utilisé pour le cas classique. Le slicing s'appuie sur les dépendances de contrôle et de données, auxquelles viennent s'ajouter les dépendances d'assertion qui associent les instructions aux assertions qui les protègent. Ces dépendances d'assertion permettent de s'assurer que l'hypothèse de protection faite plus haut reste valide dans les slices. Le slice set et la slice sont définis comme précédemment mais en utilisant les trois relations de dépendance. Une nouvelle propriété de correction est établie dans ce cadre plus général, qui est plus faible que la propriété classique. D'après cette nouvelle propriété, la projection de l'exécution du programme initial est un préfixe de la projection de l'exécution de la slice. Bien que la propriété établie soit plus faible, elle permet tout de même de répondre aux deux questions précédentes. Si la slice contient une erreur, alors trois situations sont possibles dans le programme initial : soit la même erreur se produit, soit elle est masquée par une autre erreur qui se produit avant, soit elle est masquée par une boucle infinie qui empêche l'exécution de l'atteindre. Si, en revanche, la slice ne contient pas d'erreurs, alors le programme initial ne contient pas d'erreurs non plus, si on se restreint aux instructions que le programme initial et sa slice ont en commun. Pour obtenir une confiance élevée dans les résultats, nous les formalisons dans l'assistant de preuve Coq. Un slicer certifié pour notre mini-langage représentatif peut être extrait de ce développement Coq.

Pour généraliser ces résultats, nous nous intéressons à la première brique d'un slicer indépendant du langage : le calcul générique des dépendances de contrôle. Nous formalisons dans Coq une théorie élégante de dépendances de contrôle sur des graphes orientés finis arbitraires proposée par Danicic et al. en 2011. Sur des graphes orientés quelconques, on ne peut pas utiliser la définition simple des dépendances de contrôle utilisée dans le cadre de notre mini-langage, ni la définition classique en termes de post-dominateurs qui nécessite l'existence dans le graphe

d'un unique nœud de sortie atteignable depuis tous les autres nœuds. Danicic et al. propose une définition applicable dans ce cadre plus général. Contrairement aux deux cas cités, il ne définit pas une relation binaire sur les nœuds du graphe, mais plutôt caractérise des ensembles de nœuds qui ont de bonnes propriétés. Un ensemble de nœuds est dit clos pour les dépendances de contrôle si, pour tout nœud du graphe atteignable depuis cet ensemble, l'ensemble des nœuds atteignables en premier dans cet ensemble depuis ce nœud est au plus un singleton. Un tel ensemble serait un slice set si seules les dépendances de contrôle étaient prises en compte pour le slicing. Si l'ensemble de sommets considéré n'est pas clos, on veut pouvoir calculer le plus petit sur-ensemble clos, appelé clôture. Danicic et al. montrent qu'il faut ajouter les sommets décideurs pour cet ensemble, qui sont les derniers points de choix sur les chemins qui terminent dans cet ensemble. Ils proposent également un algorithme pour calculer la clôture d'un ensemble de sommets. Comme pour le slicer pour le mini-langage, une version certifiée de l'algorithme peut être extrait du développement Coq.

L'algorithme proposé par Danicic et al. est correct, comme en atteste sa formalisation en Coq, mais il n'est pas très optimisé, comme admis par Danicic et al. eux-mêmes. Nous proposons une amélioration de l'algorithme de Danicic et al. Nous partons du constat que cet algorithme ne tire pas avantage de sa structure itérative. Au cours d'une itération, des calculs intermédiaires sont effectués, mais seuls les résultats sont transmis à l'itération suivante alors qu'elle pourrait réutiliser une partie des résultats intermédiaires obtenus. Nous proposons donc un algorithme où d'avantage d'information est transmise entre les itérations. Cette information est enregistrée sous la forme d'un étiquetage des sommets du graphe. À chaque itération cet étiquetage est partiellement mis à jour et utilisé pour détecter de nouveaux nœuds à ajouter dans la clôture. Cet algorithme est lui aussi certifié, mais cette fois-ci à l'aide de Why3 qui est plus adapté que Coq pour la transcription d'un algorithme impératif et dont la force réside dans la possibilité d'appeler des prouveurs automatiques pour éliminer les preuves simples et se concentrer sur les preuves plus compliquées.

Pour confirmer que le nouvel algorithme proposé est bien plus rapide que l'algorithme de Danicic et al., nous les avons comparés sur des graphes générés aléatoirement. Nous avons également comparé des variantes de chaque version, en particulier la version extraite de la formalisation en Coq de l'algorithme de Danicic et al. Les expériences montrent d'une part que la version extraite de Coq est particulièrement lente, et d'autre part que notre algorithme optimisé est notablement plus rapide que l'algorithme de Danicic et al.

Contents

Remerciements **v**

Résumé étendu en français **vii**

List of Figures **xv**

List of Algorithms **xix**

1 Introduction **1**
1.1 Formal Verification 2
1.2 Program Slicing 3
1.3 Slicing for Verification 6
1.4 Contributions and Outline 7

2 Presentation of Coq and Why3 **11**
2.1 Coq 11
2.1.1 Presentation 11
2.1.2 Proof of Correctness of an Insertion Sort 12
2.2 Why3 18
2.2.1 Presentation 18
2.2.2 Proof of Correctness of an Insertion Sort 19

3 Notations **25**
3.1 Program and Trajectory Notations 25
3.1.1 Programs 25
3.1.2 Trajectories 25
3.1.3 Finite Syntactic Paths 28
3.1.4 Projections 28
3.2 Graph Notations 29

4 Background: Static Backward Slicing on a WHILE Language 33
4.1 Presentation of the WHILE Language . . . 34
4.1.1 Syntax . . . 34
4.1.2 Semantics . . . 36
4.2 Dependence-Based Program Slicing on the WHILE Language . . . 40
4.2.1 Control Dependence . . . 42
4.2.2 Data Dependence . . . 43
4.2.3 Slice Set . . . 47
4.2.4 Quotient . . . 48
4.2.5 Static Backward Slicing . . . 56
4.3 Soundness Property of Program Slicing . . . 56
4.3.1 Projections . . . 56
4.3.2 Soundness Theorem . . . 60

5 Justification of Program Slicing for Verification on a Representative Language 65
5.1 Presentation of the WHILE Language with Errors . . . 66
5.1.1 Syntax . . . 66
5.1.2 Semantics . . . 71
5.2 Dependence-Based Program Slicing on the WHILE Language with Assertions . . . 73
5.2.1 Control Dependence . . . 75
5.2.2 Data Dependence . . . 76
5.2.3 Assertion Dependence . . . 77
5.2.4 Slice Set . . . 77
5.2.5 Quotient . . . 78
5.2.6 Static Backward Slicing . . . 80
5.2.7 Illustrating Examples . . . 81
5.3 Soundness Property of Relaxed Slicing . . . 85
5.3.1 Projections . . . 85
5.3.2 Soundness Theorem . . . 87
5.4 Verification on Relaxed Slices . . . 90
5.5 Remarks about the Coq Formalization . . . 93
5.5.1 Overview . . . 93
5.5.2 WHILE language with Errors . . . 99
5.5.3 Formalization of States . . . 100
5.5.4 No Use of Coinductive Types . . . 100
5.5.5 Non-Uniqueness of Labels . . . 103
5.5.6 Formalization of Data Dependencies . . . 103
5.6 Related Work . . . 104
5.6.1 Debugging and Dynamic Slicing . . . 104

5.6.2 Slicing and Non-Termination 104
5.6.3 Slicing in the Presence of Errors 108
5.6.4 Certified Slicing 109

6 Formalization of Weak Control-Closure in Coq 111
6.1 State of the Art about Control Dependence 112
6.1.1 Structured Control Flow 112
6.1.2 Unstructured Control Flow Using Control Flow Graphs . . . 112
6.1.3 Unstructured Control Flow on Arbitrary Graphs 118
6.2 Weak Control-Closed Sets 120
6.3 Weak Control-Closure 124
6.4 Link with Classic Control Dependence 135
6.5 Danicic's Algorithm for Computing Weak Control-Closure 136
6.5.1 Critical Edge 137
6.5.2 Definition and Proof of Correctness of Danicic's Algorithm . 139
6.6 Remarks about the Coq Formalization 141
6.6.1 Coq Formalization 141
6.6.2 No Use of Coinductive Types 147
6.6.3 No Generic Graph Library 147
6.6.4 Advantage of a Proof Assistant for Graph Theoretic Proofs . 147
6.6.5 Termination of Danicic's Algorithm 147

7 Design of a New Algorithm Computing Weak Control-Closure 149
7.1 General Idea of the Optimization 150
7.2 Informal Description of the New Algorithm 155
7.3 Formal Definition of the New Algorithm 161
7.3.1 Function `propagate` 162
7.3.2 Function `confirm` 163
7.3.3 Function `main` 165

8 Proof of Correctness of the New Algorithm 169
8.1 Contracts of Graph Operations 170
8.2 Proof of Correctness of `propagate` 171
8.3 Proof of Correctness of `confirm` 182
8.4 Proof of Correctness of `main` 182
8.5 Remarks about the Why3 Formalization 197
8.5.1 Overview of the Development 197
8.5.2 Proof Effort 203
8.5.3 Particularities of the Why3 Formalization 204
8.5.4 Extraction into OCaml 207

9 Experimental Comparison of Danicic's and the New Algorithms 209
9.1 Methodology . . . 209
9.1.1 Remarks about the Implementations . . . 209
9.1.2 Description of the Testing Procedure . . . 211
9.2 Presentation of the Implementations . . . 212
9.2.1 Variants of Danicic's Algorithm . . . 212
9.2.2 Variants of the Optimized Algorithm . . . 215
9.3 Results . . . 215
9.3.1 Comparison between the Coq Extraction and a Naive Danicic Implementation . . . 216
9.3.2 Impact of the Optimizations on the Naive Danicic Implementation . . . 217
9.3.3 Comparison of Naive and Smart Danicic Implementations . 219
9.3.4 Impact of the Optimizations on the Smart Danicic Implementation . . . 219
9.3.5 Impact of the Reachability Tests on a Smart Danicic Implementation . . . 222
9.3.6 Comparison between Danicic's and the Optimized Algorithms 223
9.3.7 Comparison between the Implementations of the Optimized Algorithm . . . 223
9.3.8 Impact of the Number of Edges on the Experiments . . . 226

10 Conclusion 229
10.1 Summary . . . 229
10.2 Perspectives . . . 230
10.2.1 Detection of a Wider Class of Errors . . . 230
10.2.2 A Certified Verification Chain . . . 231
10.2.3 Strong Control Dependence . . . 231
10.2.4 Further Experiments . . . 231
10.2.5 A More Optimized Algorithm for Arbitrary Control Dependence . . . 232
10.2.6 A Certified Generic Slicer . . . 233

Bibliography 235

Index 245

List of Figures

1.1 Schematic illustration of slicing applied to program understanding . 5

2.1 Definitions of natural numbers and lists in the standard library . . . 13
2.2 Definition of the sorting algorithm in Coq 14
2.3 Definition of the notion of sortedness in Coq 15
2.4 Proof that `insertion_sort` returns a sorted list 17
2.5 Extraction of `insertion_sort` into OCaml 18
2.6 Definition of the sorting algorithm in Why3 20
2.7 Schematic relation between (`a at Loop`) and `a` 21

3.1 Example graph G 29

4.1 Syntax of the WHILE language 34
4.2 Original program p 36
4.3 Example trajectory of program p of Figure 4.2 41
4.4 The original program p and its slice with respect to line 8 48
4.5 Examples ((a) and (b)) and counter-example ((c)) of quotients of p (see Figure 4.2) 50
4.6 Example trajectory of program q of Figure 4.4b 57
4.7 A program and its slice with respect to line 5 58
4.8 Example trajectories of programs of Figure 4.7 58
4.9 Projection of $\mathcal{T}[\![p]\!]\sigma$ (cf. Figure 4.3) on $S = \{2, 3, 5, 6, 8\}$ 61

5.1 Examples of statements with their protecting assertions 67
5.2 Syntax of the WHILE language 67
5.3 Two examples illustrating the use of labels in assertions 68
5.4 Program p computing in two ways the average of the values of an array `a` of size `N`, provided that only values at indices multiple of `k` are not zero 70
5.5 Example of a finite trajectory ending without error using program p of Figure 5.4 74
5.6 Example of abnormal termination using program p of Figure 5.4 . . 74

5.7 Example of infinite trajectory using program p of Figure 5.4 75
5.8 The original program p and its slice q_1 with respect to line 18 . . . 79
5.9 The original program p and its slice q_2 with respect to line 20 . . . 82
5.10 Errors ($\lightning$), non-termination ($\circlearrowleft$) and normal termination (—) of programs p (of Figure 5.4), q_1 (of Figure 5.8b) and q_2 (of Figure 5.9b) for some inputs. 83
5.11 Schematic behaviors of programs p (cf. Figure 5.4), q_1 (cf. Figure 5.8b) and q_2 (cf. Figure 5.9b) for inputs $\sigma_1, \ldots, \sigma_5$. 86
5.12 Language definition 93
5.13 Implementation of assertion dependence in Coq (cf. Definition 5.6) 95
5.14 Implementation of control dependence in Coq (cf. Definition 5.2) . 96
5.15 Implementation of data dependence in Coq (cf. Definition 5.5) . . . 96
5.16 Soundness theorem (cf. Theorem 5.1) formalized in Coq 98
5.17 Program p of Figure 4.2 in the syntax accepted by the extracted slicer 99
5.18 Output of the extracted slicer on the program of Figure 5.17 100
5.19 Definition of coinductive lists in Coq 101
5.20 Examples of coinductive lists 101
5.21 Tentative `CoFixpoint` definitions on infinite lists 102

6.1 Control flow graph G of the program of Figure 4.2 113
6.2 An example program with nested conditionals and its CFG 116
6.3 Example graph G_0, with $V_0' = \{u_1, u_3\}$ 120
6.4 Example graph G_0 annotated with the set of observable vertices with respect to $V_0' = \{u_1, u_3\}$ 122
6.5 Example graph G_0, with $V_0' = \{u_1, u_3\}$, and u_6 highlighted as a V_0'-weakly deciding vertex 126
6.6 Schematic representation of the configuration of Lemma 6.3 128
6.7 If π_2 contains some cycles, we can remove them 128
6.8 Transformation of the problem when π_2 intersects π_{21} and π_{22} . . . 130
6.9 Highlighting of the two V'-paths in the case where π_1 intersects first π_{21} and then π_{22} 130
6.10 Highlighting of the two V'-paths in the case where π_1 and π_{22} are disjoint 130
6.11 Schematic representation of the configuration of Property 6.4 131
6.12 First sub-case of the proof of Property 6.4 131
6.13 Second sub-case of the proof of Property 6.4 132
6.14 Second sub-case of the proof of Property 6.4 133
6.15 Control flow graph G of program p of Figure 4.2, with slicing criterion $V_1' = \{\texttt{enter}, u_8, \texttt{exit}\}$ 136
6.16 Control flow graph G of program p of Figure 4.2, with slicing criterion $V_{12}' = \{\texttt{enter}, u_2, u_5, u_6, u_8, \texttt{exit}\}$ 137

6.17 Optimized Danicic's algorithm applied on G_0 (cf. Figure 6.3) with $V_0' = \{u_1, u_3\}$ . . . 142
6.18 Abstract specification of graphs . . . 143
6.19 Abstract specification of finite graphs . . . 143
6.20 Hierarchy of functors . . . 144
6.21 Definition of weakly control-closed set in Coq . . . 145
6.22 Formulation of Property 6.4 in Coq ("53" refers to [DBH+11, Lemma 53]) . . . 145
6.23 Formulation of Theorem 6.1 in Coq ("54" refers to [DBH+11, Theorem 54]) . . . 146
6.24 Formulation of the correctness of Danicic's algorithm in Coq ("61" refers to [DBH+11, Algorithm 61]) . . . 146

7.1 Execution of the algorithm on an example graph. Initially, $W = \{u, v\}$. . . . 154
7.2 Execution of the algorithm on a variant of the graph shown in Figure 7.1. Initially, $W = \{u, v\}$. . . . 157
7.3 The optimized algorithm applied on G_0 (cf. Figure 6.3) with $V_0' = \{u_1, u_3\}$ . . . 159

8.1 Definitions of the invariants and assertions annotating the body of `propagate` (G, W, obs, u, v) (cf. Algorithm 8.6) . . . 175
8.2 Definitions of the invariants and assertions annotating the body of function `main` (G, V') (cf. Algorithm 8.8) . . . 186
8.3 The three possible configurations in the proof of $(\mathbf{I_7})$ . . . 192
8.4 The modeling of graphs in the Why3 formalization . . . 198
8.5 Definition of the set of V'-weakly deciding nodes in Why3 . . . 200
8.6 Some properties about `wd` . . . 200
8.7 Excerpt of the definition of `propagate` . . . 201
8.8 Definition of `main` in the Why3 development, with annotations removed . . . 202
8.9 Proof results for the preliminary definitions . . . 204
8.10 Proof results for the algorithm . . . 204
8.11 Implementation of `confirm` in Why3, with its annotations . . . 206

9.1 Comparison between the Coq extraction and a naive implementation in OCaml of Danicic's algorithm . . . 216
9.2 Comparison between two naive OCaml implementations of Danicic's algorithm, with and without optimization 1 . . . 217
9.3 Comparison between two naive OCaml implementations of Danicic's algorithm, with and without optimization 2 . . . 218

9.4 Comparison between two naive OCaml implementations of Danicic's algorithm, one with optimization 1 and one with optimizations 1 and 2 . . . 219
9.5 Comparison between the naive and smart OCaml implementations of Danicic's algorithm . . . 220
9.6 Comparison between two smart OCaml implementations of Danicic's algorithm, with and without optimization 1 . . . 221
9.7 Comparison between two smart OCaml implementations of Danicic's algorithm, with and without optimization 2 . . . 221
9.8 Comparison between two smart OCaml implementations of Danicic's algorithm, one with optimization 1 and one with optimizations 1 and 2 . . . 222
9.9 Comparison between two smart OCaml implementations of Danicic's algorithm with both optimizations, with or without reachability checks . . . 223
9.10 Comparison between the smart OCaml implementation of Danicic's algorithm with both optimizations and the optimized algorithm . . . 224
9.11 Comparison between the manual extraction from WhyML and the OCaml implementation of the optimized algorithm . . . 225
9.12 Comparison between the implementation with a second traversal and the standard implementation of the optimized algorithm . . . 225
9.13 Comparison of the smart implementation of Danicic's algorithm on graphs with $e/v = 2$ and $e/v = 1.5$ . . . 226
9.14 Comparison of the OCaml implementation of the optimized algorithm on graphs with $e/v = 2$ and $e/v = 1.5$ . . . 227

List of Algorithms

6.1 Danicic's algorithm for computing weak control-closure 140

7.1 Function `propagate` (G, W, obs, u, v) 164
7.2 Function `confirm` (G, obs, u, u') 165
7.3 Function `main` (G, V') 167

8.1 Contract of function `choose` (V) 170
8.2 Contract of function `pred` (G, u) 171
8.3 Contract of function `succ` (G, u) 171
8.4 Contract of function `out_degree` (G, u) 171
8.5 Contract of function `propagate` (G, W, obs, u, v) 172
8.6 Annotated body of `propagate` (G, W, obs, u, v) (cf. Algorithm 7.1) . . 174
8.7 Function `confirm` (G, obs, u, u') with annotations 183
8.8 Function `main` (G, V') with annotations 185

Chapter 1

Introduction

Nowadays, we are surrounded by software in our everyday lives. It is in our computers, in our tablets and in our smartphones. On these devices, we run a wide variety of programs, some provided by the manufacturer, some recommended by third parties, such as antivirus programs, and some that we choose to install ourselves to take advantage of the functionalities they offer. Using such complex systems, it is not rare to experience bugs. Does this mean that developers refuse to spend time on ensuring that their programs run correctly? Actually, this is not pure laziness from the developers and can be explained.

First, as presented above, programs are run on a wide variety of platforms which all have their specificities. Writing a program that behaves well on every hardware and on every operating system is really a hard task. When, moreover, programs rely on the interactions with other programs, it increases the probability of the occurrence of a bug.

Second, one can argue that, if developers follow a guide of good practices, they can avoid introducing a lot of bugs in their developments. This includes respecting some coding rules, having a good test suite and taking into account bugs reported by users using tools like bug tracking systems. The remaining bugs may be considered too rare by developers to be worth spending more time on detecting and correcting them. This allows them to focus on the functionalities of their software, whose presence will be far more visible than the presence of the few bugs that were not eliminated.

Third, a lot of the programs that we use are or contain free software, which is often written fully or partly by volunteers who can argue that, since they work freely, they are not forced to guarantee anything about the quality of their software. This is even written clearly in the header template of one the most famous free license, the GNU GPL license [FSF07]:

> This program is distributed in the hope that it will be useful, but

> WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Fourth, is the presence of a few minor bugs so annoying? If it concerns non-critical software, such as a multimedia player, it can be worth benefiting from its functionalities at the cost of a few bugs. If movies are shown correctly most of the time, it can be acceptable that sometimes the sound is not of good quality or even that the window closes without notice.

But if the presence of bugs can be tolerated in non-critical software, it is clearly not acceptable in critical software, when a large sum of money or human lives are at stake, since the consequences are quickly catastrophic. This kind of software can be found for instance in nuclear power plants and aircraft systems. To give an idea of the possible consequences of a bug in a critical system, the first Ariane 5 rocket, which exploded subsequent to a software bug [Lio96], was estimated at hundreds of millions of dollars. For this kind of software, it is thus important to detect and eliminate each and every bug.

To ensure the absence of bugs in software, we can make use of techniques called formal verification methods.

1.1 Formal Verification

Formal verification denotes the use of formal methods, techniques using mathematical representations of objects to mathematically reason about them, for software verification, the activity of proving that a program respects a given specification, or at least satisfies some key properties. One property of interest is the absence of runtime errors, such as divisions by zero and invalid memory accesses, that could prevent the normal execution of the program. Formal methods are particularly adapted to manipulate software, since programs are written in programming languages which are formal languages with a precise semantics.

Well-known examples of formal verification techniques include:

- Model checking [CES86]: model checking combines abstraction and exhaustive space exploration to show that a program respects a given specification.
- Symbolic execution [Kin76]: the program is executed with symbolic values, branching execution on conditional statements.
- Abstract interpretation [CC77]: the program is executed, but instead of connecting variables to concrete values, they are connected to abstractions that over-approximate the concrete values they could take in the concrete

executions. For example, an integer variable can be connected to its sign or to an interval containing the possible values it could take.

- Deductive verification [Hoa69]: a program is proved to satisfy a specification using techniques *à la* Hoare.

Note that the four techniques presented all manipulate the program without concretely executing it. Such techniques are called static analyses.

Formal verification methods can be costly. Their cost can increase greatly with the size of the program under study. Reducing the size of the program under certain conditions can help apply certain techniques that would be too costly otherwise.

1.2 Program Slicing

Program slicing is a method allowing to extract from a program a simpler program that has the same behavior with respect to a given criterion, called the slicing criterion. Typically, the slicing criterion is a statement of the program, and the slicing removes the instructions that have no impact on that statement. The resulting program is called the program slice.

Usually, a slice has to be a valid program. In particular, it has to be executable. However, some works in the literature define slices as subsets of statements of the original program. Such slices may be non-executable. This is discussed e.g. in [Tip95]. In this thesis, though, we consider only executable slices.

The original version introduced by Mark Weiser in 1981 [Wei81] is now called static backward slicing, where "static" means that the program slice must preserve the behavior of the initial program for all possible inputs, and "backward" means that the preserved statements are those that impact directly or indirectly the slicing criterion, since such statements are selected in a backward search from the criterion.

The opposite notions to "static" and "backward" exist, and are informally described below.

- In contrast with static slicing, dynamic slicing [KL88] computes a program slice valid for a given input instead of all possible inputs. Dynamic slicing can thus be more precise than static slicing.

- In contrast with backward slicing, forward slicing [BC85, RB89] preserves the statements impacted by the slicing criterion (selected in a forward search), instead of those that impact the slicing criterion.

The classifications static/dynamic and backward/forward are orthogonal. There exist the four variants (e.g. [Ven91]): backward static, forward static, backward dynamic and forward dynamic slicing.

A lot of other flavors of program slicing have been proposed and explored. Here is a non-exhaustive list.

- Quasi-static slicing [Ven91] lies between static and dynamic slicing. Some subset of the input space is considered, but is not restricted to a singleton like dynamic slicing.

- Conditioned [CCL98], precondition-based [LCYK01], postcondition-based [CH96], specification-based [LCYK01] and assertion-based slicing [BdCHP12] are various forms of program slicing that use parts of contracts as slicing criterion.

- Amorphous slicing [HBD03]. Most of the variants of slicing allows only to remove statements (or replacing them with no-op statements). In this kind of slicing, other semantic-preserving transformations are allowed.

- Observation-based slicing [BGH+14], also called ORBS, computes the slice differently from the other variants. Instead of first determining which statements have to be removed and then computing the slice, ORBS proposes to first remove some statements from the original program and then checks whether the resulting program still compiles and preserves the behavior of the original program on some inputs. If this is the case, the program is said to be a slice of the original program. The strength of ORBS is that it can slice programs written in multiple languages.

- Abstract slicing [MZ17] uses as slicing criterion a property of some data instead of its precise value, allowing to produce smaller slices.

- Specialization slicing [AHJR14] proposes to revisit slicing using ideas taken from the field of partial evaluation.

While applying slicing to a given language requires to handle the specific features of the language, some works (e.g. [FRT95, WZ07, WLS09]) observe that the concept of slicing is independent of the underlying language, and propose language-independent definitions.

Program slicing has proved to be useful in a lot of areas. It has been proposed for program understanding [KR98, HBD03], software maintenance [GL91], debugging [ADS93, KNNI02], testing [Bin98], program integration [BHR95] and software metric [PKJ06].

(a) Initial program (b) Slice with respect to instruction `l`

Figure 1.1 – Schematic illustration of slicing applied to program understanding

To give a better intuition on slicing and illustrate one of its uses, let us explain informally how it is used to help program understanding. Consider a program with an instruction of interest denoted `l`. This program is schematically represented in Figure 1.1a. Instruction `l` is surrounded by many other instructions, so that it is rather difficult to understand its purpose in the program. To detect the statements that are needed to execute `l` and remove the other ones, we apply backward slicing on the program with respect to instruction `l`. The resulting slice is represented in Figure 1.1b. The structure of the slice reveals that only $\mathtt{l}_1$ and $\mathtt{l}_2$ are needed to understand the context in which `l` is executed. We can study the three instructions in isolation in the slice, and, since backward slicing preserved all the instructions impacting `l`, we know that they have the same interaction in the initial program. Note that if we are interested in the statements impacted by `l` rather than those impacting `l`, we can apply forward slicing instead of backward slicing.

Program slicing is not a purely theoretic domain. There exist implementations of slicers for real programming languages. Some of them are rather on the research side (for instance, the Frama-C platform [KKP+15] has a slicing plug-in for the C language; Indus [RH07] is a slicer for concurrent Java programs with an Eclipse-based GUI), while some are available in the industry (for instance, Codesurfer [ART03] for C and C++).

Several surveys were conducted on program slicing. The best-known is probably the one written by Frank Tip [Tip95] in 1995, but it is becoming a bit old. Other surveys include [BG96, BH04, Kri05, XQZ+05, Sil12].

In this thesis, we are interested in static backward program slicing that produces executable slices. In the remainder of this document, unless stated otherwise, the word “slicing” will refer to this version of slicing.

One goal of slicing is to produce the smallest possible slice, i.e. the slice with the minimal number of statements. But computing the minimal slice is undecidable [Wei84]. Slices are thus conservative over-approximations of the minimal slice.

More precisely, the statements preserved in the slice are all the statements that *may* have an influence on the slicing criterion.

Traditionally, in static backward slicing, the impact of a statement on another one is modeled using control dependencies and data dependencies [FOW87]. In a given program, statement s_2 is control dependent on statement s_1 if the execution of s_1 influences whether s_2 is executed. Statement s_2 is data dependent on statement s_1 if a variable read in s_2 may have been last assigned at s_1. The slice contains all the statements on which the slicing criterion is directly and indirectly control or data dependent. We present this approach in much more detail in Chapter 4.

1.3 Slicing for Verification

Static backward slicing appears like a good candidate for reducing the size of a program and applying formal verification techniques that otherwise would be too expensive. But the use of program slicing in the context of program verification requires a solid theoretical foundation that has not been clearly established. This means that either we refuse to use slicing for verification because of this lack of foundation or we still decide to use it at the cost of a reduced confidence in the results.

Below we detail two techniques that successfully apply slicing to help verification.

Chebaro et al. [CKGJ12] describe a method called SANTE, that takes advantage of static backward slicing to help detect runtime errors in a given C program p.

The SANTE method has three steps. In a first step, a value analysis of p is conducted using abstract interpretation that detects possible runtime errors (alarms) such as divisions by zero, out-of-bounds array accesses and some cases of invalid pointers. This analysis detects all such errors, but can produce false positives. The objective of the following steps is to classify the alarms as true errors or as false positives. In a second step, p is sliced into one or several slices. The resulting slices $p_1, \ldots, p_n$ each contain a subset of alarms. The third step applies dynamic analysis to each slice p_i and tries to find, for each alarm in p_i, an input that triggers the associated error. This analysis classifies each alarm, or returns unknown if it does not succeed in classifying it.

The SANTE method was implemented in the FRAMA-C platform [KKP+15]. First experiments [CKGJ12, CCK+14] showed that the approach of SANTE is effective. Especially, combining value analysis and slicing allows the verification to be on average 43% faster when applying SANTE with slicing than when applying the dynamic analysis directly on the initial program (without even a value

analysis).

Slabý et al. [SST12] also propose to use program slicing in a verification process. More precisely, their goal is to prove that a program verifies a property described by a finite state machine. The architecture of their technique is similar to that of SANTE. First, the program is instrumented to reflect the behavior of the finite state machine in the program. Second, the program is sliced such that the slice has the same behavior as the original program with respect to the instrumentation. Third, the slice is symbolically executed to determine if an error of the instrumentation can be reached.

Early experiments on C programs show that slicing removes on average 60% of the code of the instrumented programs. Slabý et al. implemented their technique in a tool called SYMBIOTIC that operates on C programs [SST13].

SANTE and SYMBIOTIC are therefore two approaches taking advantage of program slicing in the context of verification. However, the soundness of these approaches was not clearly established by the authors. In this thesis, we aim to provide a justification of the use of slicing in these techniques to provide a high level of confidence in their results.

1.4 Contributions and Outline

This thesis brings the required theoretical foundation to support the use of program slicing in the context of verification. We focus on the detection of errors determined by the program state such as runtime errors. Especially, we answer the following questions about the link between the presence or the absence of errors in a program and in its slices:

- If we prove the absence of errors in a slice, what can be said of the original program?
- If an error is detected in a slice, does the same error occur in the original program?

We model our problem using a simple representative imperative language with potential runtime errors and non-terminating loops. Allowing the presence of non-terminating loops in addition to runtime errors is important since they occur in realistic programs and cannot be excluded before running some verification on them.

In this general context, the classic soundness property of program slicing does not hold. Indeed, since non-terminating loops and runtime errors can both prevent the execution of the following statements, removing them by slicing breaks the

equivalence of behaviors between the original program and the slice. The straightforward solution would be to add more dependencies than in the standard case. Instead, we introduce *relaxed slicing* that keeps few dependencies and establish an appropriate soundness property. This soundness property, weaker than the classic one, still allows us to establish the link between the presence or the absence of errors in the original program and in its slices, and thus answer the two questions asked above.

To ensure the highest confidence in the results as possible, the whole work (the definition of relaxed slicing, the soundness property and the consequences for verification) is formalized in the Coq proof assistant for a language representative for our purpose. A certified slicer for this language can be automatically extracted into OCaml from this formalization.

To apply our results on richer languages, and still be able to extract a certified implementation, we propose to formalize a generic, i.e. language-independent, slicer. The first step in this direction is the formalization of an algorithm computing generic control dependence.

We formalize a generalization of control dependence on arbitrary finite directed graphs [DBH+11] taken from the literature in the Coq proof assistant. This includes both the theoretical concepts, an iterative algorithm to compute from a subset of vertices the smallest superset closed under control dependence, and a proof that this algorithm is correct.

The formalized algorithm is iterative but does not fully take advantage of its iterative nature to share information between iterations. We propose a new iterative algorithm that shares intermediate results between iterations and is thus more efficient than the original one. This new algorithm being more complex, its proof of correctness relies on more complicated invariants than the original one. We choose the Why3 proof system to formalize it, to take advantage of the automatic provers that it can call. The formalization includes all the necessary concepts, the algorithm and a proof of its correctness.

To compare experimentally the original algorithm and our optimized version, we implement both in OCaml and run them on thousands of randomly generated graphs with up to thousands of vertices. These experiments show that our new algorithm clearly outperforms the original one.

This thesis is structured as follows.

Chapter 2 presents the two main proof systems that we use in this thesis: the Coq proof assistant and the Why3 platform. In particular, it illustrates how to prove the correctness of an insertion sort in these systems.

Chapter 3 introduces some notations that are used in the rest of the thesis. It is written to serve as a reference.

Chapter 4 presents classic static backward slicing on a simple imperative language. It recalls the main definitions of program slicing and illustrates them on this language.

The following chapters present the main contributions of this thesis. I was the main contributor of these achievements and the main author of the papers [LKL16a, LKL16b, LKL18a, LKL18b, LKL18c].

- Chapter 5 reuses and extends the definitions of Chapter 4 for the same language augmented with errors. On this language, it justifies the use of slicing for verification. Then it presents the related work on slicing for verification.

  In particular, this chapter introduces:

  - the notion of relaxed slicing for structured programs with possible errors and non-termination, which keeps fewer statements than it would be necessary to satisfy the classic soundness property of slicing;
  - a new soundness property for relaxed slicing using a trajectory-based denotational semantics;
  - a characterization of verification results, such as absence or presence of errors, obtained for a relaxed slice, in terms of the initial program, that constitutes a theoretical foundation for conducting verification on slices;
  - the formalization and proof of correctness of relaxed slicing in Coq for a representative language, from which a certified slicer for the considered language can be extracted.

  This work has been published in [LKL16a, LKL16b]. An extended version of it has been published in [LKL18a]. The Coq code discussed in this chapter is available online [Léc16].

- Chapter 6 presents the formalization in Coq of the theory of control dependence on finite directed graphs of [DBH+11], together with the associated algorithm and the proof of its correctness. It first reviews the domain of control dependence, then presents and illustrates some concepts introduced in [DBH+11], the algorithm and the proof of its correctness.

  This work has been published in [LKL18b, LKL18c]. The Coq code presented in this chapter is available online [Léc18].

- Chapter 7 presents and illustrates a new algorithm optimizing Danicic's algorithm by taking benefit from preserving some intermediate results between iterations.

  This work has been published in [LKL18b, LKL18c].

- Chapter 8 describes in detail the mechanized correctness proof of this new algorithm in the Why3 tool that also includes a formalization of the necessary concepts.

  Part of this work has been published in [LKL18b, LKL18c]. The Why3 code presented in this chapter is available online [Léc18].

- Chapter 9 presents the implementation of both algorithms in OCaml, their evaluation on random graphs and a comparison of their execution times.

  Part of this work has been published in [LKL18b, LKL18c]. The OCaml code of the implementations is available online [Léc18].

Chapter 10 concludes and presents some perspectives.

Chapter 2

Presentation of Coq and Why3

In this chapter, we present two programs that allow to write computer-aided proofs and that are used to machine-check the main results of this thesis (see Chapters 5, 6 and 8): the Coq proof assistant [Coq17, BC04] and the Why3 proof platform [Why18, FP13].

The main difference between the two programs is their level of automation. Coq is an interactive theorem prover, and thus it requires human interaction, at least to some extent. On the contrary, the strength of Why3 is that it can call automatic solvers as backends to prove the goals automatically.

This chapter presents Coq in Section 2.1 and Why3 in Section 2.2. In each chapter, we briefly introduce the tool, before illustrating it on the proof of an insertion sort.

2.1 Coq

2.1.1 Presentation

Coq is a proof assistant, that has been successfully used to produce both theoretical results, such as the four-color theorem [Gon08], and more practical ones, such as the formalization of a C compiler called CompCert [Ler09].

From a user point of view, Coq allows to:

- define objects (e.g. lists of natural numbers) and functions (e.g. a sorting function) in a pure functional language;
- define properties about these objects (e.g. the sortedness of a list) in higher-order logic;

- state theorems (e.g. the list returned by the call of the sorting function on an arbitrary list is sorted);
- prove these theorems.

As a programming language, Coq is a rather unusual one. From a user perspective, there exist two languages. The first language, called Gallina, can be used for the three first points listed above: defining objects, their properties and stating theorems. The second language is used for the fourth point: proving the theorems. When we want to prove a theorem, what we get is a list of hypotheses and a conclusion to be proved. This is called a goal. This second language defines instructions, called tactics, that allow to reduce a goal to a simpler one, such that if we prove the second one, we have a proof of the first one. After applying several tactics, we can reduce the initial goal to a list of basic goals that can be proved in a straightforward manner, which proves the theorem.

While at first sight there seem to be two really different languages, the Coq kernel which is the trusted base of Coq, understands only one, Gallina. Coq indeed relies on the Curry-Howard correspondence, which means that the logic is encoded in the typing system. More precisely, certain types are seen as formulas, and objects of these types are seen as proofs of these formulas. Checking that an object is a proof of a theorem comes down to checking that it has the right type. Tactics are just user-friendly instructions that allow the user to manipulate the goals in an intuitive manner. But behind the scenes, a Gallina term is built progressively. At the end of the proof, the type-checker verifies that the term built by the tactics is indeed a proof of the theorem, i.e. has the correct type.

For the interested reader, the theory behind Coq is called the Calculus of Inductive Constructions. We do not present this theory in this chapter. Rather, we focus on a presentation of Coq as a tool. The next section presents the concrete syntax and some basic mechanisms in the context of the proof of correctness of an insertion sort.

2.1.2 Proof of Correctness of an Insertion Sort

In this section, we show how to prove an insertion sort on lists of natural numbers in Coq.

First, we recall the definitions of the standard types that we need: natural numbers and lists. Figure 2.1 presents the definitions of these two types in the Coq standard library. Both are introduced by keyword `Inductive` and end with a dot ". ". This is a general rule about commands in Gallina. They begin with a keyword and end with a dot. This kind of definition with `Inductive` defines inductive types. An inductive type is introduced with some constructors that are

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.

Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A.
```

Figure 2.1 – Definitions of natural numbers and lists in the standard library

the only way to construct objects of this type. In the case of `nat`, a natural number (of type `nat`) is either `O`, interpreted as zero, or `S n`, interpreted as the successor of `n`, i.e. `n`+1. Likewise, a list (of type `list`) is either the empty list (`nil`) or the addition of an element to a list (`cons`).

Three remarks can be made about these definitions. First, the types are defined in the sorts `Set` and `Type`. This means that they are considered as datatypes (as opposed to logic types). In this context, `Inductive` introduces classic algebraic datatypes. Second, one can observe that the definition `list` is parametrized by `A` of type `Type`. This defines `list` as a polymorphic type. For instance, `list nat` is the type of lists of natural numbers; `list` (`list nat`) is the type of lists of lists of natural numbers. Third, constructors, and more generally functions in Coq, are curried. This means that they receive their arguments one by one instead of receiving a tuple once. For instance, the type assigned to `cons` in the definition of `list` is `A → list A → list A`. This means that it has two arguments: one of type `A`, the other of type `list A`. We can provide them one after the other. For instance, `cons` 2 is a function that adds the number 2 at the front of a list. `cons` 2 `nil` is a list with 2 as its single element.

We can now define our sorting algorithm. It is given in Figure 2.2. It is made up of two functions. In terms of notations:

- [] denotes the empty list;
- [`n`] denotes a list with a single element, `n`;
- " :: " is an infix notation for `cons`; `m` :: `l`' can be read as `cons m l`';
- "<=?" is the less than or equal operator on `nat`; it returns a Boolean of type `bool`.

The first function, `insert`, inserts a number into a list. `insert n l` searches for the first element in `l` that is less than or equal to `n` and inserts `n` just before

```
Fixpoint insert (n : nat) (l : list nat) :=
    match l with
    | []  ⇒ [n]
    | m:: l'  ⇒ if n <=? m then n::l else m::insert n l'
    end.

Fixpoint insertion_sort (l : list nat) :=
    match l with
    | []  ⇒ []
    | m:: l'  ⇒ insert m (insertion_sort l')
    end.
```

Figure 2.2 – Definition of the sorting algorithm in Coq

that element. `insert` is written to insert correctly the number in an already sorted list, so that the resulting list is also sorted. This definition uses two features that have not been presented yet: recursion and pattern-matching.

`insert` is indeed a recursive function. It is defined using keyword `Fixpoint` that introduces recursive functions in Coq. The particularity of recursive functions in Coq is that they must terminate. This is for consistency reasons, since non-terminating functions could lead in certain circumstances to a proof of `False`, from which we could prove anything. Coq can automatically detect that some function terminates, if one argument is structurally smaller in the recursive calls than in the definition. Indeed, this guarantees that there cannot exist an infinite chain of recursive calls. In the case of `insert`, `l` is the decreasing argument, since the argument of the recursive call in the case where `l` is non-empty is `l'` which is strictly smaller than `l`. This is shown by the message produced by Coq when it reads the definition of `insert`:

"`insert` is recursively defined (decreasing on 2nd argument)."

If we try to define a non-terminating function, Coq refuses the definition. For example, if we replace `l'` in the recursive call with `l`, Coq fails with the message:

"Error: Cannot guess decreasing argument of fix."

Coq proposes several mechanisms to define functions that do not respect this criterion but still terminate. In this relax setting, one can define a custom well-founded order and prove that some argument of the recursive function is smaller in the recursive calls than in the definition with respect to this well-founded order. This is briefly discussed in Section 6.6.5.

The other feature that the definition of `insert` uses is pattern-matching that allows to perform case-analysis on an object that has an inductive type. This is

```
Inductive Sorted : list nat → Prop :=
| Sorted_nil : Sorted []
| Sorted_single : forall n, Sorted [n]
| Sorted_cons : forall n m l, n <= m → Sorted (m::l) → Sorted (n::m::l).
```

Figure 2.3 – Definition of the notion of sortedness in Coq

the case of list `l`, that is either the empty list ([]) or a non-empty list (`m:: l'`).

The main function of this sorting algorithm is `insertion_sort`. If the list given as an argument is empty, then the empty list is returned. It the list is non-empty, the tail of the list is recursively sorted by `insertion_sort l'` and the first element is inserted at the right position using insert.

Now that we have defined the sorting algorithm, we want to prove that it is correct. We first need to define what it means to be sorted. This is defined using an inductive predicate. The exact definition is shown in Figure 2.3. Actually, a more general notion is defined in the Coq standard library, but we do not use it for pedagogical reasons.

`Sorted` operates on lists of type `list nat`, i.e. on lists of natural numbers, and returns in `Prop` that is the sort of the logical formulas. Due to the Curry-Howard correspondence, we recognize familiar concepts that we have already used in `Type`. First, the definition of `Sorted` uses the same construction as the definitions of `nat` and `list`. It is also defined inductively. In the context of logical formulas, inductive definitions introduces predicates *à la* Prolog. Second, in `Sorted_cons`, one can observe that the arrow "→ " that is used to denote a function type is also used to denote implication. This is because, in the Curry-Howard correspondence, an implication `P → Q` is really the type of functions transforming proofs of `P` into proofs of `Q`. Like the type of functions, the implications are curried. Instead of writing `P ∧ Q → R` (i.e. "if `P` and `Q` then `R`), we rather write `P → Q → R` (i.e. "if `P` then if `Q` then `R`").

By definition of `Sorted`, a list of natural numbers is sorted if and only if:

- it is empty (`Sorted_nil`), or
- it contains a single element (`Sorted_single`), or
- it contains at least two elements, the first one is less than or equal to the second one, and the tail of the list is sorted (`Sorted_cons`).

Note that in the definition of `Sorted`, we use the notation "<=", while in the definition of `insert` we used the notation "<=?". Actually, "<=?" is the Boolean

operator that can be used to write the functions. "<=" is the logical operator which can be used in the logical world. What we can prove is that "<=?" is a correct implementation of "<=", i.e. that for two natural numbers `n` and `m`, `n <=? m` returns `true` if and only if `n <= m`. This is proved in the Coq standard library:

```
Lemma leb_le : forall n m : nat, (n <=? m) = true ↔ n <= m.
```

What we want to prove about `insertion_sort` is that it sorts correctly. In other words, we want to prove that the returned list is sorted and has the same elements as the input list. For the sake of concision, we prove only that the returned list is sorted.[1] In the language of Coq, this can be written like this:

```
Theorem insertion_sort_sorted : forall (l : list nat),
  Sorted (insertion_sort l).
```

To prove this result about `insertion_sort`, we need to establish some results about `insert`. Again for the sake of concision, we admit that `insert` inserts correctly an element in a sorted list. In Coq, this corresponds to the following axiom:

```
Axiom insert_sorted : forall (n : nat) (l : list nat), Sorted l
                      → Sorted (insert n l).
```

With this axiom, the proof `insertion_sort_sorted` is simple. We give it in Figure 2.4. It is introduced with the keyword `Proof`. This is a simple proof by induction. The first tactic, `intros`, introduces `l` in the context. This is the equivalent of the English sentence: "Let `l` be an arbitrary list of natural numbers". The second tactic, `induction l`, performs induction on `l`. This produces two goals, each introduced by a dash.

- The first goal is the base case. We need to prove that the theorem holds if `l` is the empty list, i.e.

  `Sorted (insertion_sort [])`

  The tactic `simpl` simplifies `insertion_sort []` into `[]` . Thus, what we need to prove is:

  `Sorted []`

  This is exactly `Sorted_nil`, that we use with tactic `apply`.

[1] The full proof is available on http://perso.ecp.fr/~lechenetjc/tools/.

```
Theorem insertion_sort_correct : forall (l : list nat),
    Sorted (insertion_sort l).
Proof.
    intros. induction l.
    - simpl. apply Sorted_nil.
    - simpl. apply insert_sorted. assumption.
Qed.
```

Figure 2.4 – Proof that `insertion_sort` returns a sorted list

- The second goal is the induction step. We need to prove that, given a number `n` and provided that `Sorted (insertion_sort l)` (the induction hypothesis), we have:

  `Sorted (insertion_sort (n::l)`

  We first use `simpl` to compute `insertion_sort (n::l)`. This gives:

  `Sorted (insert n (insertion_sort l)`

  This is where we can apply the axiom `insert_sorted`. We obtain:

  `Sorted (insertion_sort l)`

  This is the induction hypothesis. We conclude with `assumption` that proves the goal if the conclusion is one of the hypotheses.

We end the proof with `Qed`. Coq verifies that the proof term built by the tactics is correct and defines `insertion_sort_sorted`. Theorem `insertion_sort_sorted` thus becomes an established fact that can be used in the rest of the development.

One last feature that we want to highlight is the extraction mechanism, that allows to extract any Coq function (in `Type`) into a functional language such as OCaml, Haskell or Scheme. In this thesis, we use several times the extraction mechanism into OCaml (see Sections 5.5.1 and 6.6.1). For example, we can extract function `insertion_sort`. The command is:

```
Extraction insertion_sort.
```

The OCaml code that is produced is shown in Figure 2.5. This allows to produce an executable version of function `insertion_sort`. If we trust the extraction mechanism, since we have proved that `insertion_sort` is correct in Coq, we can describe the OCaml version as a certified implementation of a insertion sort.

```
(** val insertion_sort : nat list -> nat list **)

let rec insertion_sort = function
| Nil -> Nil
| Cons (m, l') -> insert m (insertion_sort l')
```

Figure 2.5 – Extraction of `insertion_sort` into OCaml

Obviously, this example has covered only a small portion of the principles of Coq. This should be sufficient, though, to understand the high-level structure of the proofs. The interested reader can refer to the official website [Coq17] for more resources about Coq. Two of them [PCG+15, BC04] are particularly appropriate for beginners.

2.2 Why3

2.2.1 Presentation

Why3 is a proof platform that can be used at least for two purposes:

- It can be used as an interface with backend provers. In this use case, Why3 is only seen as a logical language with convenient access to multiple provers.
- Why3 can also be used as a tool to model algorithms. They can be written in a language called WhyML. This language allows both to write programs in a language similar to OCaml and to write logic in a first-order logic with inductive predicates. Our use of Why3 falls into this second category.

To prove programs written in WhyML, we use deductive verification *à la* Hoare. Programs are given specifications in the form of contracts listing some preconditions and postconditions. To prove that these contracts are valid, we annotate the function bodies. We use three types of annotation:

- We annotate each loop with loop invariants that abstract the behavior of the loop body. If we prove that the invariants hold when entering the loop and are preserved by a loop iteration, we obtain that they also hold when leaving the loop.
- We annotate each loop with an expression called a variant. If we prove that, after each iteration, this variant is smaller than before the iteration with respect to a given well-founded order, we show that the loop cannot run an

infinite number of times and thus terminates. In the simple cases, the variant is a non-negative integer expression and the order used is the standard less than order on integers.

- We add assertions at specific points in the program. They state that some intermediate fact holds at the point where they are located. They are not strictly needed, but help the proof.

We present some elements of the syntax of WhyML on the proof of an insertion sort.

2.2.2 Proof of Correctness of an Insertion Sort

In this section, we show how to prove an insertion sort on arrays of integers in Why3. Contrary to the insertion sort that we have presented for Coq in Section 2.1.2, we do not operate on lists but on arrays, and the sorting is performed in-place. It is quite possible to prove the same sorting algorithm on lists,[2] but we prefer to present an imperative program rather than a functional one. This resembles more the kind of programs we prove with Why3 in this thesis (see Section 8.5).

Like in the previous section, for the sake of concision, we do not prove that the sorting function preserves the contents of the array. We only prove that the array at the end of the function is sorted.[3]

We first define a logical predicate `sorted` to define what is a sorted array:

```
predicate sorted (a : array int) (i : int) (j : int) =
  forall k l. i <= k <= l < j -> a[k] <= a[l]
```

Given an array of integers `a` and two integers `i` and `j`, `sorted a i j` states that `a` is sorted between `i` (included) and `j` (excluded). We use a definition of sortedness that compares each pair of indices in the array. We prefer this definition to the definition in Section 2.1.2 that defines sortedness by comparing only consecutive indices. Since the less than order on natural numbers is transitive, both definitions are equivalent. But the one that we use in this section is manipulated more easily by SMT provers. In particular, it does not require to perform inductive reasoning.

The definition of annotated algorithm is given in Figure 2.6. It has one post-condition introduced by **`ensures`** on line 2, which states that at after the call to `insertion_sort`, the array `a` is sorted between the indices `0` and `Array.length a`, i.e. is fully sorted.

[2]The proof of the insertion sort on lists is presented in the gallery of examples of Why3 (http://toccata.lri.fr/gallery/insertion_sort_list.en.html).

[3]The full proof is available on http://perso.ecp.fr/~lechenetjc/tools/.

```
let insertion_sort (a : array int)
  ensures { sorted a 0 (Array.length a) }
=
  for i = 0 to Array.length a - 1 do
    invariant { sorted a 0 i }
    let tmp = a[i] in
    let j = ref i in
    label Loop in
    while !j > 0 && a[!j-1] > tmp do
    invariant { 0 <= !j <= i }
    invariant { forall k. 0 <= k < !j -> a[k] = a[k] at Loop }
    invariant { forall k. !j+1 <= k < i+1 -> a[k] = a[k-1] at Loop }
    invariant { forall k. !j+1 <= k < i+1 -> tmp <= a[k] }
    variant { !j }
    a[!j] <- a[!j-1];
    j := !j -1
  done;
  a[!j] <- tmp
  done
```

Figure 2.6 – Definition of the sorting algorithm in Why3

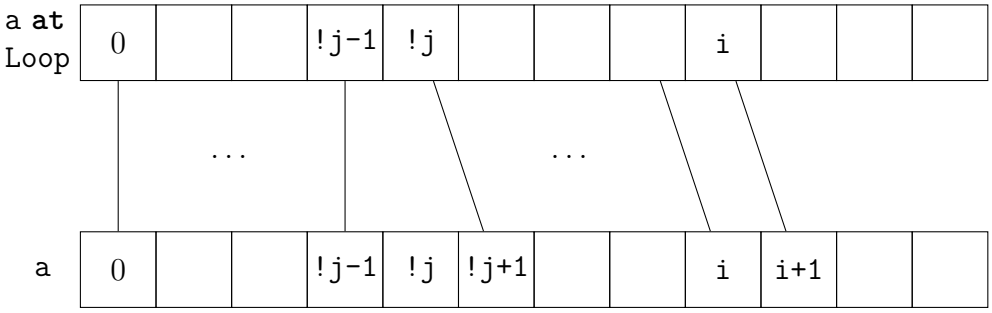


Figure 2.7 – Schematic relation between (`a at Loop`) and `a`

The body of the function performs a standard insertion sort on `a`. We sort gradually the array, storing the sorted sub-array in the left part of array `a`. To process the next element, we insert it at the right index in the sorted sub-array. For that, we shift all the elements of the sorted sub-array that are greater than the new element one cell to the right. This leaves a free cell where we insert the new element.

More precisely, we process each value in `a` one after the other in the loop on line 4. Let us consider iteration $(\texttt{i}+1)^{\text{th}}$. First, we copy the value of `a` at index `i` in variable `tmp` (on line 5). Then, we traverse the array backwards from `i` with the loop on line 9. The current inspected index is stored in reference `j`. References are the simplest way to have mutable variables in WhyML and OCaml. While `!j` (the value contained in `j`) is greater than zero, and the value at index `!j - 1` is greater than `a[i]`, we copy `a[!j - 1]` into `a[!j]` (line 15) and decrement `j` (line 16). If we reach the beginning of `a` or if we find before some `j` such that `a[!j-1]` is greater than `a[i]`, we insert `a[i]` into `a` at index `!j` (line 18).

To prove the correctness of the function, we add loop invariants and variants to the loops on line 4 and 9.

The invariant of the outer loop on line 4 is simple. It states that the sub-array between indices `0` and `i` is sorted.

The invariants of the inner loop on line 9 are a bit more complex. The first one (on line 10) describes the range of values taken by `!j`.

The next two invariants relate the contents of `a` after some iterations of the inner loop and the contents of `a` when entering the inner loop. For that, we introduce a label named `Loop` on line 8 to refer to the point of the program just before the loop. In the invariants, we use the expression **`at`** `Loop` to refer to the value of a variable at the point of the program designated by `Loop`, i.e. its value before entering the inner loop. After some iterations, `a` can be described based on `a` **`at`** `Loop`, as illustrated in Figure 2.7. The left part (between `0` and `!j`) is identical to the left part of `a` **`at`** `Loop`, the right part (between `!j+1` and `i+1`) is

the right part of `a at Loop` moved one cell to the right. The value of `a` at index `!j` is not specified.

The last invariant allows us to prove that we insert `a[i]` at the right index. It states that if we have not inserted it yet when testing index `!j-1`, all the shifted elements, i.e. those at indices between `!j+1` and `i+1`, are greater than `tmp`, i.e. `a[i]`.

Let us briefly explain how the invariants of the inner loop can prove that the insertion performed at line 18 is correct, i.e. guarantees the sortedness of the left part of `a`. When leaving the inner loop, we know that either `!j` is equal to zero, or `a[!j-1]` is less than or equal to `tmp`. If `!j` is equal to zero, then by the third invariant of the inner loop, the section of `a` between indices `1` and `i+1` is sorted, and by the fourth invariant, `tmp` is smaller than all the elements in this section. By inserting `tmp` at index `0` on line 18, we get an array `a` that is sorted between indices `0` and `i+1`. If `a[!j-1]` is less than or equal to `tmp`, then by the second invariant, the left section (between indices `0` and `!j`) is sorted, and all its elements are smaller than or equal to `a[!j-1]`, and thus are smaller than or equal to `tmp`. By the third invariant, the right section (between indices `!j+1` and `i+1`) is sorted too, and all its elements are greater then or equal to `tmp` by the fourth invariant. By inserting `tmp` at index `!j`, we get an array `a` is sorted between indices `0` and `i+1`.

In Why3, we do not implement this reasoning manually. We take advantage of one of the provers it can call. This includes the following SMT solvers: Alt-Ergo [Alt16, BCCL08], CVC4 [CVC17, BCD+11] and Z3 [Z16, dMB08], and a theorem prover called E [E17, Sch13]. In the case where some goal is too difficult for the automatic provers, Why3 can also call Coq to prove it manually. The strength of Why3 is that it can use a different prover for each goal. For example, we can use Alt-Ergo to prove most of the goals, then CVC4 on the remaining ones, and then Coq if some goals are still unproved. Section 8.5 shows a non-trivial use case where the five provers that we have just mentioned are used together. For this simple use case, though, all the goals are proved automatically by Alt-Ergo in 0.12s.

The insertion sort on arrays is also formalized and proved in the gallery of examples of Why3.[4] More generally, a rich variety of resources is available on the Why3 website [Why18], including the manual, the gallery of examples and some links to publications. Note however that the version of Why3 that we use in this thesis is the development version.[5] The reader might experience some differences between what is presented in this thesis and what is available online

[4] http://toccata.lri.fr/gallery/insertion_sort.en.html
[5] https://gitlab.inria.fr/why3/why3

which is compatible with the current release (0.88.3). The main ideas are the same, though.

Chapter 3

Notations

This chapter presents the few notations that are used in this thesis. This is just a compilation of them that can be used as a reference. It can be skipped on a first reading.

3.1 Program and Trajectory Notations

3.1.1 Programs

We have three types whose exact definitions are left abstract:

- labels (often denoted l);
- arithmetic expressions (often denoted e);
- Boolean expressions (often denoted b).

Using these three types, we can give an explicit definition of the syntax *Stmt* of statements (see Chapter 4 and Chapter 5). Statements are often denoted s.

The syntax *Prog* of programs is defined as a (possible empty) list of statements. Formally:

$$Prog \quad ::= \quad Stmt^*$$

A program is either empty (denoted λ), or a finite sequence of statements (denoted $s_1; \ldots; s_k$, where $s_1, \ldots, s_k$ are statements). Programs are often denoted p.

3.1.2 Trajectories

States. States are mappings associating integer values to variables. They are often denoted σ. The set of states is denoted Σ. Two notations are used to manipulate states:

- A state can be defined explicitly, giving the extensive list of bindings. The state $\sigma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ associates, for any i such that $1 \leq i \leq n$, the value v_i to the variable x_i.

- A state can be defined using another state, by adding one or more bindings to it. If one of the added bindings affects a variable which already has a value in the original state, it overrides the previous value. If σ is a state, $\sigma[x \leftarrow v]$ adds the binding $x \mapsto v$ to the state σ. We can update multiple variables at the same time, if these variables are pairwise distinct. For example,

  $$\sigma[x \leftarrow v, y \leftarrow w]$$

  adds the bindings $x \mapsto v$ and $y \mapsto w$ to σ.

  Given the variables x, y and z and the state $\sigma = \{x \mapsto 0, y \mapsto 1\}$,

  - $\sigma[x \leftarrow 2] = \{x \mapsto 2, y \mapsto 1\}$;
  - $\sigma[x \leftarrow 0] = \{x \mapsto 0, y \mapsto 1\}$;
  - $\sigma[z \leftarrow 2] = \{x \mapsto 0, y \mapsto 1, z \mapsto 2\}$.

Trajectories. Trajectories, also named traces, are finite or countably infinite lists of (label, state)-pairs. They are often denoted T and are represented using angle brackets ("$\langle$" and "$\rangle$") enclosing them.

- $T = \langle\rangle$ is the empty trajectory.

- $T = \langle(l_1, \sigma_1) \ldots (l_k, \sigma_k)\rangle$ is a finite trajectory of length k, where for any i such that $1 \leq i \leq k$, (l_i, σ_i) is the i-th element of T

- $T = \langle(l_1, \sigma_1) \ldots (l_i, \sigma_i) \ldots\rangle$ is a possibly infinite trajectory.

Concatenation of trajectories. $\oplus$ is the concatenation operator over trajectories. When applied on two trajectories T_1 and T_2, if T_1 is finite and does not end with the error state (see Chapter 5), $T_1 \oplus T_2$ is defined classically as the trajectory obtained by adding the elements of T_2 at the end of the elements of T_1. Formally, given a finite trajectory

$$T_1 = \langle(l_1, \sigma_1) \ldots (l_k, \sigma_k)\rangle$$

that does not end with the error state, and

$$T_2 = \langle(m_1, \tau_1) \ldots (m_i, \tau_i) \ldots\rangle$$

the concatenation of T_1 and T_2 is defined as follows:

$$T_1 \oplus T_2 = \langle (l_1, \sigma_1) \dots (l_k, \sigma_k)(m_1, \tau_1) \dots (m_i, \tau_i) \dots \rangle$$

In the case where T_1 is infinite or ends with the error state, the concatenation $T_1 \oplus T_2$ is equal to T_1. Moreover, $\oplus$ is lazy and ignore T_2 in that case. This means that we can write some equations including concatenations of the form $T_1 \oplus T_2$, with T_2 not defined properly when T_1 is infinite or ends with the error state. This is used in Chapter 4 and Chapter 5.

Last state of a trajectory. For a finite trajectory T and a state $\sigma \in \Sigma$, we define $LS_\sigma(T)$ as the last state of T (i.e. the state component of its last element) if $T \neq \langle\,\rangle$, and σ otherwise. Formally, we define a function with two arguments:

$$LS_(_) : State \times Trajectory \to State$$

that verifies:

- $LS_\sigma(\langle\rangle) = \sigma$;
- $LS_\sigma(\langle (l_1, \sigma_1) \dots (l_n, \sigma_n) \rangle) = \sigma_n$, if $n > 0$;
- $LS_\sigma(T)$ is undefined if T is infinite.

Choice operator. Given a Boolean value v and two trajectories T_1 and T_2, we define $(v \to T_1, T_2)$ as

$$(v \to T_1, T_2) = \begin{cases} T_1 \text{ if } v = \mathit{True} \\ T_2 \text{ if } v = \mathit{False} \end{cases}$$

Prefix of trajectories. Given a natural number n and a trajectory

$$T = \langle (l_1, \sigma_1) \dots (l_i, \sigma_i) \dots \rangle$$

of length at least n (or infinite), the prefix of T of length n, denoted $T^{(n)}$, is defined classically as the finite trajectory containing the n first elements of T. Formally,

$$T^{(n)} = \langle (l_1, \sigma_1) \dots (l_n, \sigma_n) \rangle$$

3.1.3 Finite Syntactic Paths

Finite syntactic paths. Finite syntactic paths are finite lists of labels. They can be obtained from finite trajectories by removing the state component of each element. They are often denoted π.

- $\pi = \langle\rangle$ is the empty finite syntactic path.
- $\pi = \langle l_1, \ldots, l_k \rangle$ is a finite syntactic path of length k, where for any i such that $1 \leq i \leq k$, l_i is the i-th element of π.

For instance, to a trajectory $T = \langle (l_1, \sigma_1)(l_2, \sigma_2)(l_3, \sigma_3) \rangle$ of length 3, we can associate the finite syntactic path $\pi = \langle l_1, l_2, l_3 \rangle$ of the same length.

Concatenation of finite syntactic paths. The concatenation of finite syntactic paths is defined classically, and uses the same notation $\oplus$ as the concatenation of trajectories. This notation is again extended to support the concatenation of sets of finite syntactic paths (often denoted P). The concatenation of two sets is classically defined as the set of the concatenations. Formally, given P_1 and P_2 two sets of finite syntactic paths,

$$P_1 \oplus P_2 = \{\pi_1 \oplus \pi_2 | \pi_1 \in P_1, \pi_2 \in P_2\}$$

Given a set of finite syntactic paths P and a natural number n, we define P^n as the set of the concatenations of n finite syntactic paths, each taken from P. Formally, we define P^n recursively as follows:

$$P^n = \begin{cases} \{\langle\rangle\} \text{ if } n = 0 \\ \{\pi_1 \oplus \pi_2 | \pi_1 \in P^{n-1}, \pi_2 \in P\} \text{ otherwise} \end{cases}$$

P^* is the set of concatenations of any number of finite syntactic paths taken from P. It is defined as the union of all the P^n:

$$P = \bigcup_{n \in \mathbb{N}} P^n$$

3.1.4 Projections

Projections are operations that filter some information from the objects on which they are applied.

Projection of a state. The projection of a state $\sigma \in \Sigma$ to a set of variables V, denoted $\sigma{\downarrow}V$, is the restriction of σ to V.

Given the variables x, y and z and the state $\sigma = \{x \mapsto 0, y \mapsto 1\}$,

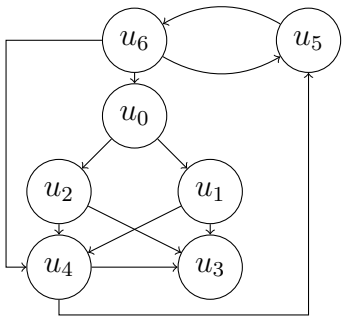


Figure 3.1 – Example graph G

- $\sigma{\downarrow}\{x\} = \{x \mapsto 0\}$;
- $\sigma{\downarrow}\{x, y\} = \{x \mapsto 0, y \mapsto 1\}$;
- $\sigma{\downarrow}\{x, z\} = \{x \mapsto 0\}$.

Projection of a trajectory. The projection of a one-element sequence $\langle (l, \sigma) \rangle$ to a set of labels L, denoted $\langle (l, \sigma) \rangle{\downarrow}L$, is defined as follows:

$$\langle (l, \sigma) \rangle{\downarrow}L = \begin{cases} \langle (l, \sigma{\downarrow}\mathsf{used}(l)) \rangle & \text{if } l \in L, \\ \langle \, \rangle & \text{otherwise} \end{cases}$$

where $\mathsf{used}(l)$ is informally the set of variables occurring in the statement of label l (see Section 4.2.2 for the exact definition).

The projection of a trajectory $T = \langle (l_1, \sigma_1) \ldots (l_k, \sigma_k) \ldots \rangle$ to L, denoted $\mathsf{Proj}_L(T)$, is defined element-wise:

$$\mathsf{Proj}_L(T) = \langle (l_1, \sigma_1) \rangle{\downarrow}L \oplus \ldots \oplus \langle (l_k, \sigma_k) \rangle{\downarrow}L \oplus \ldots$$

3.2 Graph Notations

In this section, we introduce some notations about finite directed graphs. Throughout the section, $G = (V, E)$ denotes a graph with set of vertices V and set of edges E, and V' denotes a subset of V.

We illustrate the notations presented on this section on the example graph shown in Figure 3.1.

Path. A path in G between two vertices u and v is a sequence of vertices

$$u_0 = u, \ldots, u_n = v \ (n \geq 0)$$

such that

$$\forall i, 0 \leq i < n \implies (u_i, u_{i+1}) \in E$$

The existence of a path between two vertices u and v is denoted $u \xrightarrow{path} v$.

In the graph of Figure 3.1, $u_1 \xrightarrow{path} u_3$, since u_1, u_3 is a path in G; $u_2 \xrightarrow{path} u_0$, since u_2, u_4, u_5, u_6, u_0 is a path in G; $u_3 \xrightarrow{path} u_3$, since u_3 is a (trivial) path in G.

Reachable nodes. The set of vertices in G reachable from V' is the set of vertices that are reachable from at least one vertex in V'. It is denoted $\mathsf{R}_G(V')$.

In the example graph, $\mathsf{R}_G(\{u_3\}) = \{u_3\}$, $\mathsf{R}_G(\{u_2, u_4\}) = V$.

V'-disjoint path and V'-path. A V'-disjoint path in G is a path whose vertices are not in V' except the last one that may or may not be in V'. The existence of such a path between two vertices u and v is denoted $u \xrightarrow{V'-disjoint} v$.

A V'-path is a V'-disjoint path whose last vertex is in V'. If a V'-path exists between two vertices u and v, we write $u \xrightarrow{V'-path} v$.

In the example graph (cf. Figure 3.1), for $V' = \{u_1, u_3\}$, $u_6 \xrightarrow{V'-disjoint} u_2$ (by V'-disjoint path u_6, u_0, u_2) and $u_6 \xrightarrow{V'-disjoint} u_1$ (by V'-disjoint path u_6, u_0, u_1). Since u_2 is not in V', it is false that $u_6 \xrightarrow{V'-path} u_2$. Since u_1 is in V', we have not only $u_6 \xrightarrow{V'-disjoint} u_1$, but also $u_6 \xrightarrow{V'-path} u_1$.

Observable set. Given a vertex $u \in V$, the set of nodes that u can reach using a V'-path is called the set of observable vertices from u in V' and denoted $\mathsf{obs}_G(u, V')$.

In the example graph (cf. Figure 3.1), for $V' = \{u_1, u_3\}$, $u_6 \xrightarrow{V'-path} u_1$, thus $u_1 \in \mathsf{obs}_G(u_6, V')$. More precisely, $\mathsf{obs}_G(u_6, V') = V'$. u_1 is the only vertex that u_1 can reach using a V'-path (the trivial path from u_1 is a V'-path), thus $\mathsf{obs}_G(u_1, V') = \{u_1\}$.

V'-weakly committing vertex. A vertex $u \in V$ is V'-weakly committing if $obs_G(u, V')$ contains at most one element. The set of V'-weakly committing vertices is denoted $\mathsf{WC}_G(V')$.

In the graph of Figure 3.1, $\mathsf{WC}_G(\{u_1, u_3\}) = \{u_1, u_3\}$, $\mathsf{WC}_G(\{u_5, u_6\}) = V$ and $\mathsf{WC}_G(V) = V$.

V'-weakly deciding vertex. A vertex $u \in V$ is V'-weakly deciding if there exist two V'-paths from u that share no vertex except u. The set of V'-weakly deciding vertices is denoted $\mathsf{WD}_G(V')$.

In the example graph (cf. Figure 3.1), for $V' = \{u_1, u_3\}$, $u_6 \in \mathsf{WD}_G(V')$ (by V'-paths u_6, u_0, u_1 and u_6, u_4, u_3). Likewise, u_0, u_2 and u_4 are V'-weakly deciding

vertices. On the contrary, $u_5 \notin \mathsf{WD}_G(V')$ since every V'-path from u_5 has u_6 as second vertex. Therefore, $\mathsf{WD}_G(V') = \{u_0, u_2, u_4, u_6\}$.

Weak control-closure. The weak control-closure of V' is the smallest superset W of V' such that all the vertices reachable from W are W-weakly committing. It is denoted $\mathsf{WCC}_G(V')$. We can prove (see Theorem 6.1) that $\mathsf{WCC}_G(V') = V' \cup (\mathsf{WD}_G(V') \cap \mathsf{R}_G(V'))$.

In the example graph (cf. Figure 3.1), for $V' = \{u_1, u_3\}$, we have $\mathsf{R}_G(V') = V$ and $\mathsf{WD}_G(V') = \{u_0, u_2, u_4, u_6\}$. Thus,

$$\mathsf{WCC}_G(V') = \{u_0, u_1, u_2, u_3, u_4, u_6\} = V \setminus \{u_5\}$$

Chapter 4

Background: Static Backward Slicing on a WHILE Language

As stated in Chapter 1, in this thesis we focus on the original version of program slicing: static backward slicing. The inputs of static backward slicing are a program p and a slicing criterion C. The slicing criterion may have several forms. Initially, in Weiser's work [Wei81, Wei82], it was a pair (l, V), where l is an instruction in p and V a subset of the variables occurring in p. It can also be only an instruction l (e.g. [OO84]). This second form is a particular case of the first one where V contains all the variables occurring in instruction l. We use in this thesis another form of slicing criterion: a list of instructions (like e.g. [Amt08]). This last form asks to preserve every statement contained in the slicing criterion C.

In this chapter, we describe and illustrate the classic definition of static backward slicing (as defined in [Wei81, RY89]) on a simple WHILE language (e.g. [NNH99]) using the formalism of [BBD+10]. This chapter is the basis of Chapter 5, which will use a slightly extended version of this language.

This WHILE language is an imperative language with arithmetic expressions, assignments, conditionals and loops. We use a trajectory-based semantics (as in e.g. [BBD+10]). We define static backward slicing in a classic way using control and data dependence relations. Using these relations, we can construct from the slicing criterion the set of instructions that have an influence on the slicing criterion and thus have to be preserved in the slice, called the slice set (e.g. [RAB+07]). Using this slice set and the initial program, we can finally construct the program slice.

This chapter is organized as follows. Section 4.1 presents the syntax and the semantics of the WHILE language. Next, Section 4.2 describes the two dependence relations on this language and uses them to define program slicing on this WHILE language. Section 4.3 states the soundness property that connects the semantics of the initial program and its slices.

$$
\begin{array}{lcl}
\mathit{Prog} & ::= & \mathit{Stmt}^* \\
\mathit{Stmt} & ::= & l : \texttt{skip} \mid \\
 & & l : x = e \mid \\
 & & \texttt{if}\ (l : b)\ \mathit{Prog}\ \texttt{else}\ \mathit{Prog} \mid \\
 & & \texttt{while}\ (l : b)\ \mathit{Prog} \\
\text{where} & & \\
 & & l, l' : \text{label} \\
 & & e : \text{expression} \\
 & & b : \text{Boolean expression}
\end{array}
$$

Figure 4.1 – Syntax of the WHILE language

Each result presented in this chapter has a counterpart in the next chapter for the slightly extended language. Moreover, the results of the next chapter are mechanically proved in Coq (see Section 5.5). Strictly speaking, the results of this chapter are not proved in Coq, but, since the languages and the proved statements are similar in both chapters, we abusively mark a proof in this chapter as mechanized in Coq when its equivalent in the next chapter is mechanized in Coq.

4.1 Presentation of the WHILE Language

4.1.1 Syntax

We first describe the structure of the WHILE language. It is a simple imperative language, with conditionals and loops. Its precise syntax is described in Figure 4.1.

A program (*Prog*) is a possibly empty list of statements (*Stmt*). The empty list is denoted λ, and the list separator is ";". A program p can be decomposed into the list of its statements $p = s_1; \ldots; s_n$, with $n \geq 0$.

Defining programs as lists of statements allows us to consider empty programs, which is convenient to define empty branches, and to manipulate statements without being hampered by the associativity of ";" for statements. For example, any non-empty program p can be decomposed into its first statement s followed by the rest of the program q, written $p = s;\ q$. This will be systematically used in the recursive definitions and the proofs by induction.

There are four kinds of statements. These are basic blocks found in most imperative languages. Their semantics is briefly described here, but is given in more detail in Section 4.1.2.

- the l : **skip** statement, which, as its name suggests, does nothing
- the assignment $l : x = e$, where x is assigned the evaluation of e
- the conditional **if** $(l : b)$ p **else** q, evaluating p or q depending on the evaluation of b
- the loop **while** $(l : b)$ p, evaluating p as long as the evaluation of b is true

Figure 4.1 does not specify the precedences of the constructions of the languages. For completeness, we give to **else** and **while** higher precedences than the sequence operator ";". For example, **while** $(l : b)$ p_1; p_2 is read as (**while** $(l : b)$ p_1); p_2. In practice, though, we systematically add curly brackets ("{" and "}") around the branches of the conditions and the bodies of the loops when needed to avoid any ambiguity.

Each statement is given a label. Labels are identifiers that give names to the statements of a program. We assume that the labels of any given program are distinct, so that a label uniquely identifies a statement in this program. The exact type of labels is left abstract. In practice, line numbers are a simple solution to associate labels to statements. In the following of this section, we systematically refer to a statement using its label. Given a program p, the set of its labels is denoted $L(p)$. The slicing criterion C is a set of labels of p, i.e. $C \subseteq L(p)$. Function L is illustrated below on the example of Figure 4.2.

Like labels, the type of expressions is left abstract. They are usual Boolean and arithmetic expressions. The important property of our expressions is that they are pure, i.e. they have no side effect and thus cannot modify variables directly. We assume that expressions come with a function vars such that for any Boolean or arithmetic expression e, vars(e) returns the set of variables occurring in e. Function vars is illustrated below on Figure 4.2.

We now introduce the running example that is used in this section to illustrate the concepts of program slicing. It is a simple program p, written in the WHILE language, represented in Figure 4.2, which takes as inputs two positive integer variables a and b and returns 1 if b divides a and 0 otherwise. Actually, our language does not contain a "return" statement, thus we need to model the "return" instruction. In this program, we chose to store the result of the program

```
1: quo = 0;
2: r = a;
   while (3: b <= r) {
4:    quo = quo + 1;
5:    r = r - b;
   }
   if (6: r != 0) {
7:    res = 0;
   } else {
8:    res = 1;
   }
```

Figure 4.2 – Original program p

in a variable named `res`. The first part of the program (instructions 1–5) computes the euclidean division of `a` by `b`. The second part (instructions 6–8) tests the remainder `r` of the division to determine if `b` divides `a`.

Program p contains labels 1, 2, 3, 4, 5, 6, 7, 8, therefore, by definition, $L(p) = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

We illustrate vars on certain expressions occurring in p:

- $\mathsf{vars}(\texttt{quo + 1}) = \{\texttt{quo}\}$
- $\mathsf{vars}(\texttt{r != 0}) = \{\texttt{r}\}$
- $\mathsf{vars}(\texttt{r - b}) = \{\texttt{r}, \texttt{b}\}$

4.1.2 Semantics

We now give the precise meaning of WHILE programs. As described above, assignments, conditions and loops have the usual semantics, and **`skip`** does nothing.

To formalize this, we use a denotational trajectory-based semantics, like in e.g. [BBD+10].

First, we need the notion of program state. In the WHILE language, it is simply a mapping from variables to values. Let Σ denote the set of program states. We use two notations to manipulate states:

- A program state can be specified explicitly, giving the exact list of variables defined and associated values. The state

  $$\sigma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$$

  associates, for any i such that $1 \leq i \leq n$, the value v_i to the variable x_i.

- A program state can be defined from another program state, adding one or more bindings to it. Given $\sigma \in \Sigma$, $\sigma[x \leftarrow v]$ is σ updated with the binding $x \mapsto v$, which means that each variable is connected to the same value as in σ except x which is connected to v. If x is not bound in σ, a binding is added in $\sigma[x \leftarrow v]$. If x is bound in σ, the binding is updated in $\sigma[x \leftarrow v]$.

  Given the variables x, y and z and the state $\sigma = \{x \mapsto 0, y \mapsto 1\}$,

  - $\sigma[x \leftarrow 2] = \{x \mapsto 2, y \mapsto 1\}$;
  - $\sigma[x \leftarrow 0] = \{x \mapsto 0, y \mapsto 1\}$;
  - $\sigma[z \leftarrow 2] = \{x \mapsto 0, y \mapsto 1, z \mapsto 2\}$.

Given a program, not all the states are interesting. Indeed, they need to be related to the program, i.e. to contain bindings about the variables present in the program. Furthermore, they need to give a value to all the variables read before being assigned in the program, to rule out the case where an expression cannot be evaluated due to undefined variables. We choose a conservative way of ensuring that the execution will not be stuck. We consider only the states that give a value to each variable of the program. We call such states *initial states* of the program.

Definition 4.1: Initial state

Given a program p, an *initial state* of p is a state that associates a value to every variable in p.

Consider program p shown in Figure 4.2. The set of variables present in p is $\{\mathtt{a}, \mathtt{b}, \mathtt{quo}, \mathtt{r}, \mathtt{res}\}$, an initial state of p must therefore define each variable in this set. For example, $\{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 20, \mathtt{r} \mapsto 200, \mathtt{res} \mapsto 5\}$ is an initial state of p. On the contrary, $\{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1\}$ is not an initial state of p, while p could be correctly executed on it.

We can now introduce informally the concept of trajectory. Trajectories are lists of (label, state)-pairs encountered during the execution of a program. More precisely, let p be a program and $\sigma \in \Sigma$ be an initial state of p. The *trajectory* of the execution of p on σ, denoted $\mathcal{T}[\![p]\!]\sigma$, is the sequence of pairs $(l_1, \sigma_1) \ldots (l_i, \sigma_i) \ldots$, where $l_1, \ldots, l_k, \ldots$ is the sequence of labels of the executed instructions, and $\sigma_i \in \Sigma$ is the state of the program *after* the execution of instruction l_i.

$\mathcal{T}$ can be seen as a (partial, since we consider only initial states) function

$$\mathcal{T} : Prog \times \Sigma \to Seq(L \times \Sigma)$$

where $Seq(L \times \Sigma)$ is the set of sequences of pairs $(l, \sigma) \in L \times \Sigma$. Trajectories can be finite if the execution terminates or (countably) infinite if the execution

does not terminate. Trajectories are denoted using enclosing angle brackets ("$\langle$" and "$\rangle$"). In particular, the empty trajectory is denoted $\langle\,\rangle$. A finite trajectory of size n is denoted $\langle(l_1, \sigma_1)\ldots(l_n, \sigma_n)\rangle$. A possibly infinite trajectory is denoted $\langle(l_1, \sigma_1)\ldots(l_i, \sigma_i)\ldots\rangle$.

Formally, $\mathcal{T}$ is defined by recursion over our WHILE language. We first need to introduce a few definitions and notations before giving this formal definition.

Let $\oplus$ be the concatenation operator over sequences. The definition of $T_1 \oplus T_2$ is standard if T_1 is finite. If T_1 is infinite, then we set $T_1 \oplus T_2 = T_1$ for any T_2 (and even if T_2 is not well-defined, in other words, $\oplus$ performs lazy evaluation of its arguments).

We also need a function that returns the last state of a finite trajectory. It takes as arguments a trajectory and a state to be returned in the case where the trajectory is empty. For a finite trajectory T and a state $\sigma \in \Sigma$, we define $LS_\sigma(T)$ as the last state of T (i.e. the state component of its last element) if $T \neq \langle\,\rangle$, and σ otherwise. Formally, we define a function with two arguments:

$$LS_(_) : State \times Trajectory \rightarrow State$$

that verifies:

- $LS_\sigma(\langle\rangle) = \sigma$;
- $LS_\sigma(\langle(l_1, \sigma_1)\ldots(l_n, \sigma_n)\rangle) = \sigma_n$, if $n > 0$;
- $LS_\sigma(T)$ is undefined if T is infinite.

We denote by $\mathcal{E}$ an evaluation function for expressions that is standard and not detailed here. For any (pure) expression e and state $\sigma \in \Sigma$ associating a value to each variable in e, $\mathcal{E}[\![e]\!]\sigma$ is the evaluation of expression e using σ to evaluate the variables present in e. For example, given the variables x and y, if $\sigma = \{x \mapsto 1, y \mapsto 0\}$, then

$$\mathcal{E}[\![2 * x + x * y]\!]\sigma = 2 * 1 + 1 * 0 = 2$$

The last notation we need is a choice operator to define the meaning of conditionals and loops. Given a Boolean value v and two trajectories T and T', we introduce the notation $(v \rightarrow T, T')$ as

$$(v \rightarrow T, T') = \begin{cases} T \text{ if } v = \mathit{True} \\ T' \text{ if } v = \mathit{False} \end{cases}$$

We can now give the recursive definition of $\mathcal{T}$ for any valid state $\sigma \in \Sigma$.

Definition 4.2

The trajectory $\mathcal{T}[\![p]\!]\sigma$ of a program p on initial state σ is recursively defined as follows:

$$
\begin{aligned}
\mathcal{T}[\![\lambda]\!]\sigma &= \langle\,\rangle && (1)\\
\mathcal{T}[\![s;\ p]\!]\sigma &= \mathcal{T}[\![s]\!]\sigma \oplus \mathcal{T}[\![p]\!](LS_\sigma(\mathcal{T}[\![s]\!]\sigma)) && (2)\\
\mathcal{T}[\![l : \texttt{skip}]\!]\sigma &= \langle (l,\sigma)\rangle && (3)\\
\mathcal{T}[\![l : x = e]\!]\sigma &= \langle (l,\sigma[x \leftarrow \mathcal{E}[\![e]\!]\sigma])\rangle && (4)\\
\mathcal{T}[\![\texttt{if}\ (l : b)\ p\ \texttt{else}\ q]\!]\sigma &= \langle (l,\sigma)\rangle \oplus (\mathcal{E}[\![b]\!]\sigma \rightarrow \mathcal{T}[\![p]\!]\sigma, \mathcal{T}[\![q]\!]\sigma) && (5)\\
\mathcal{T}[\![\texttt{while}\ (l : b)\ p]\!]\sigma &= \langle (l,\sigma)\rangle \oplus (\mathcal{E}[\![b]\!]\sigma \rightarrow && (6)\\
&\qquad \mathcal{T}[\![p]\!]\sigma \oplus \mathcal{T}[\![\texttt{while}\ (l : b)\ p]\!](LS_\sigma(\mathcal{T}[\![p]\!]\sigma)),\\
&\qquad \langle\,\rangle)
\end{aligned}
$$

The definition of $\mathcal{T}$ is adapted from the one given in [BBD+10, Section 4.1]. Due to its recursive nature, its well-definedness is questionable.[1] In particular, case (6) defines the trajectory of a loop based recursively on the trajectory of itself (on another state). Case (6) can thus be applied indefinitely if $\mathcal{E}[\![b]\!]\sigma$ is evaluated to true in each recursive call, i.e. if the considered loop runs infinitely on state σ. Actually, this infinite recursion is intentional, since infinite loops are supposed to produce countably infinite trajectories, but it requires some justification. The key point that justifies this definition is the fact that for any non-zero natural number k, the k-th element of a trajectory, when it exists, can be computed in finite time. Indeed, the evaluation of each statement (cases (3)–(6)) produces a (label, state)-pair, and the only operation used to assemble trajectories is the concatenation (cases (2), (5) and (6)) that preserves the first elements of the trace. Thus, in case of infinite traces, each element, say at rank k, of the trace is uniquely defined, since intuitively it suffices to perform k elementary execution steps of statements to get its value.

We describe each rule of Definition 4.2 hereafter.

(1) The trajectory produced by an empty program is always empty.

(2) The trajectory of a sequence $s;\ p$ on σ, where s is a statement and p a program, in the case where the trajectory T of s on σ is finite, is the concatenation of T and the trajectory of p on the last state of T if T is not

[1] In [BBD+10], this issue is not discussed. The function $\mathcal{T}$ is considered naturally well-defined.

empty and σ otherwise, which is exactly $LS_\sigma(T)$. Actually, T cannot be empty due to the definition of $\mathcal{T}$ in cases (3)–(6), thus $LS_\sigma(T)$ is the last state of T. This corresponds to executing s on σ first and then executing p on the resulting state.

If the trajectory T of s on σ is infinite, $LS_\sigma(T)$ is not defined, but we rely on the laziness of the $\oplus$ operator. In this case, since T is infinite, the second operand is ignored.

(3) **`skip`** does not modify the state on which it is executed, but still modifies the trajectory, contrary to the empty program.

(4) An assignment to a variable x updates the input state so that, after the assignment, x contains the result of the evaluation of the expression assigned to it. We use the notation introduced before to denote the overriding of the input state by the binding $x \mapsto \mathcal{E}[\![e]\!]\sigma$. An assignment to x only modifies x, by the assumption that we only have pure expressions.

(5) The definitions for a conditional and a loop both rely on the notation $(v \to T, T')$. The trajectory of a conditional on σ is the trajectory of a **`skip`** statement, followed by the trajectory of the then-branch on σ if the Boolean condition is evaluated to *True*, or the trajectory of the else-branch on σ if it is evaluated to *False*.

(6) The trajectory of a loop is the classic unrolling into successive conditionals. If the Boolean condition is evaluated to *True*, the body of the loop is evaluated once and we recursively evaluate the loop on the resulting state. Note that, here again, we rely on the laziness of $\oplus$ to obtain a correct definition even if the loop body produces an infinite trace.

Let us illustrate this concept of trajectories by using the running example shown in Figure 4.2. Let $\sigma = \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}$. By Definition 4.1, σ is an initial state of p. By using Definition 4.2, we can compute the trajectory of p on σ. It is given in Figure 4.3. As expected, it enters twice the loop of the euclidean division, then leaves it and assigns `res` to 1 since the remainder `r` is zero.

4.2 Dependence-Based Program Slicing on the WHILE Language

Now that we have properly defined the language by giving its syntax and its semantics, we can define static backward slicing for this language. We define control and data dependences, and then program slicing based on these relations.

$$
\begin{aligned}
\mathcal{T}[\![p]\!]\sigma = \langle & (1, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}) \\
& (2, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 2, \mathtt{res} \mapsto 0\}) \\
& (3, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 2, \mathtt{res} \mapsto 0\}) \\
& (4, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 1, \mathtt{r} \mapsto 2, \mathtt{res} \mapsto 0\}) \\
& (5, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 1, \mathtt{r} \mapsto 1, \mathtt{res} \mapsto 0\}) \\
& (3, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 1, \mathtt{r} \mapsto 1, \mathtt{res} \mapsto 0\}) \\
& (4, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 2, \mathtt{r} \mapsto 1, \mathtt{res} \mapsto 0\}) \\
& (5, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 2, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}) \\
& (3, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 2, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}) \\
& (6, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 2, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}) \\
& (8, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 2, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 1\}) \\
\rangle &
\end{aligned}
$$

where

$$\sigma = \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}$$

Figure 4.3 – Example trajectory of program p of Figure 4.2

Traditionally, in the literature, several intermediate objects are introduced to compute the slice. First, a program is assimilated to its control flow graph (CFG). Then, the slice is computed using dataflow equations [Wei84], or using another program representation such as the program dependence graph (PDG) [OO84, FOW87] if the program contains a unique procedure, in which case the slicing is said to be intraprocedural, or the system dependence graph (SDG) [HRB88] if the program contains multiple procedures, in which case the slicing is said to be interprocedural. For the sake of simplicity, and because the WHILE language is simple, structured and does not allow to write programs with multiple procedures, we use the more direct approach of [BBD+10]. We work directly on programs, not CFGs, and do not use any intermediate structure.

4.2.1 Control Dependence

Control dependence models the impact that statements have on the control flow of the program. Informally, statement s_2 is control dependent on statement s_1 if s_1 decides whether s_2 is executed.

The classic definition of control dependence in terms of post-domination (e.g. [FOW87], see also Section 6.1) can be reworded for this structured language in a simple way. The following formulation is based on the fact that, in this WHILE language, only conditionals and loops can introduce control dependence.

Definition 4.3: Control dependence $\mathcal{D}_c$

The control dependencies in p are defined by **`if`** and **`while`** statements in p as follows:

- For any statement **`if`** $(l : b)$ q **`else`** r and $l' \in L(q) \cup L(r)$, we define $l \xrightarrow{\mathcal{D}_c} l'$;
- For any statement **`while`** $(l : b)$ q and $l' \in L(q)$, we define $l \xrightarrow{\mathcal{D}_c} l'$.

In our example represented in Figure 4.2, instructions 4 and 5 are both control dependent on line 3 ($3 \xrightarrow{\mathcal{D}_c} 4$ and $3 \xrightarrow{\mathcal{D}_c} 5$); instructions 7 and 8 are control dependent on line 6 ($6 \xrightarrow{\mathcal{D}_c} 7$ and $6 \xrightarrow{\mathcal{D}_c} 8$).

Note that Definition 4.3 defines a transitive control dependence relation. For example, if a while loop is nested in the body of another while loop, the statements in the inner loop body are control dependent not only on the inner while condition but also on the outer while condition. With a non-transitive definition, such statements would not be directly control dependent on the outer loop condition. They would only be transitively control dependent on it, since they would be control dependent on the inner loop condition that would be itself control dependent on the

outer loop condition. In the literature (cf. Section 6.1.2), there exist both transitive definitions of control dependence and non-transitive ones. Since program slicing relies on the reflexive and transitive closure of the dependence relations (cf. Section 4.2.3), this does not really make a difference.

4.2.2 Data Dependence

Data dependence models the impact of statements on the values of variables. Informally, statement s_2 is data dependent on statement s_1 if a variable read at s_2 may have been last assigned at s_1.

Data dependence is classically defined in terms of def-use paths [ASU86]. A def-use path is a path in the CFG of a program between two statements s_1 and s_2 such that a variable v is assigned at s_1, read at s_2 and not assigned meanwhile on this path. Def-use paths can be computed using a standard dataflow analysis called reaching definitions [NNH99].

Formally, in our WHILE language, we define data dependence using finite syntactic paths, like in [BBD+10]. Finite syntactic paths are a way of modeling finite paths in the CFG without introducing the CFG. Intuitively, a finite syntactic path is an abstraction of a finite trajectory, where we keep only the labels and ignore the state components. This means that when encountering a conditional or a loop, it is always possible to choose which branch is selected or how many times the loop executes. Note in particular that given a finite trajectory, the list of the labels occurring in this trajectory is a finite syntactic path. The definition is thus very similar to the definition of trajectories.

We reuse $\oplus$, already introduced in Section 4.1.2 to denote the concatenation of traces, to denote the concatenation of paths. We extend it further to denote the concatenation of a set of paths as the set of concatenations of their elements. Formally, given two sets of finite syntactic paths P_1 and P_2,

$$P_1 \oplus P_2 = \{\pi_1 \oplus \pi_2 | \pi_1 \in P_1, \pi_2 \in P_2\}$$

We use the classic notation "$*$" to denote Kleene closure. Given a set of finite syntactic paths P and natural number n, we define P^n as the set of the concatenations of n finite syntactic paths, each taken from P. Formally, we define P^n recursively as follows:

$$P^n = \begin{cases} \{\langle\rangle\} \text{ if } n = 0 \\ \{\pi_1 \oplus \pi_2 | \pi_1 \in P^{n-1}, \pi_2 \in P\} \text{ otherwise} \end{cases}$$

P^0 contains only the empty finite syntactic path $\langle\rangle$. If $n > 0$, P^n contains the concatenations of any finite syntactic path in P^{n-1} and any finite syntactic path in P.

P^* is the set of concatenations of any number of finite syntactic paths taken from P. It is defined as the union of all the P^n:

$$P = \bigcup_{n \in \mathbb{N}} P^n$$

Definition 4.4: Finite syntactic paths

The set of finite syntactic paths $\mathcal{P}(p)$ of a program p is recursively defined as follows:

$$\begin{aligned}
\mathcal{P}(\lambda) &= \{\langle\rangle\} && (1)\\
\mathcal{P}(s;\ p) &= \mathcal{P}(s) \oplus \mathcal{P}(p) && (2)\\
\mathcal{P}(l : \texttt{skip}) &= \{\langle l\rangle\} && (3)\\
\mathcal{P}(l : x = e) &= \{\langle l\rangle\} && (4)\\
\mathcal{P}(\texttt{if}\ (l : b)\ p\ \texttt{else}\ q) &= \{\langle l\rangle\} \oplus (\mathcal{P}(p) \cup \mathcal{P}(q)) && (5)\\
\mathcal{P}(\texttt{while}\ (l : b)\ p) &= (\{l\} \oplus \mathcal{P}(p))^* \oplus \{\langle l\rangle\} && (6)
\end{aligned}$$

We describe each rule of Definition 4.4 hereafter.

(1) The empty program produces a single finite syntactic path, the empty one.

(2) A finite syntactic path of a sequence $s;\ p$ is the concatenation of a finite syntactic path of the first statement s and a finite syntactic path of the rest of the program p.

(3) A **skip** instruction produces a single finite syntactic path containing only its label.

(4) An assignment also produces a single finite syntactic path containing only its label.

(5) A finite syntactic path of a conditional of label l is the prepending of l to a finite syntactic path of the then-branch or the else-branch.

(6) A finite syntactic path of a loop is built by traversing an arbitrary number of times in the condition and the loop body, and in the condition one more time.

In our example shown in Figure 4.2, $\langle 1, 2, 3, 4, 5, 3, 6, 7 \rangle$ is a finite syntactic path of p.

We now introduce a few classic definitions to designate variables read or written at a given statement. Let l denote a label.

The set $\mathsf{def}(l)$ denotes the set of variables defined at statement l. Since only assignments define variables, $\mathsf{def}(l) = \{x\}$ if l is an assignment to variable x, and $\emptyset$ otherwise. Note that compound statements, i.e. conditionals and loops, are considered not defining any variable, even though one of their branches or theirs bodies could contain assignments.

Definition 4.5

The set $\mathsf{def}(l)$ of the variables defined at l is defined as follows:

$$\begin{array}{rcl} \mathsf{def}(l : \texttt{skip}) & = & \emptyset \\ \mathsf{def}(l : x = e) & = & \{x\} \\ \mathsf{def}(\texttt{if}\ (l : b)\ p\ \texttt{else}\ q) & = & \emptyset \\ \mathsf{def}(\texttt{while}\ (l : b)\ p) & = & \emptyset \end{array}$$

Considering program p of Figure 4.2,

$$\mathsf{def}(1) = \{\texttt{quo}\},\ \mathsf{def}(2) = \{\texttt{r}\} \text{ and } \mathsf{def}(3) = \emptyset.$$

The set $\mathsf{ref}(l)$ denotes the set of variables referenced or read at statement l. Its definition is based on function vars introduced in Section 4.1.1 which, given an expression, returns the set of variables occurring in this expression. Note that, again, compound statements are handled in a particular manner, since they are considered referencing only the variables in their Boolean condition, not the variables occurring in one of their branches or their bodies.

Definition 4.6

The set $\mathsf{ref}(l)$ of the variables referenced at l is defined as follows:

$$\begin{array}{rcl} \mathsf{ref}(l : \texttt{skip}) & = & \emptyset \\ \mathsf{ref}(l : x = e) & = & \mathsf{vars}(e) \\ \mathsf{ref}(\texttt{if}\ (l : b)\ p\ \texttt{else}\ q) & = & \mathsf{vars}(b) \\ \mathsf{ref}(\texttt{while}\ (l : b)\ p) & = & \mathsf{vars}(b) \end{array}$$

Considering program p of Figure 4.2,

$$\mathsf{ref}(1) = \emptyset,\ \mathsf{ref}(2) = \{\mathtt{a}\} \text{ and } \mathsf{ref}(3) = \{\mathtt{b}, \mathtt{r}\}$$

The set $\mathsf{used}(l)$ denotes the set of variables either defined or referenced at l. It is simply defined as the union of $\mathsf{def}(l)$ and $\mathsf{ref}(l)$.

Definition 4.7

The set $\mathsf{used}(l)$ of the variables used at l is defined as follows:

$$\mathsf{used}(l) = \mathsf{def}(l) \cup \mathsf{ref}(l)$$

Considering program p of Figure 4.2,

$$\mathsf{used}(1) = \{\mathtt{quo}\},\ \mathsf{used}(2) = \{\mathtt{r}, \mathtt{a}\} \text{ and } \mathsf{used}(3) = \{\mathtt{b}, \mathtt{r}\}$$

From finite syntactic paths on the one hand, and def and ref sets on the other hand, we can define data dependence.

Definition 4.8: Data dependence $\mathcal{D}_d$

Let l and l' be labels of a program p. We say that there is a data dependency $l \xrightarrow{\mathcal{D}_d} l'$ if $\mathsf{def}(l) \neq \emptyset$ and $\mathsf{def}(l) \subseteq \mathsf{ref}(l')$ and there exists a path $\pi = \pi_1 l \pi_2 l' \pi_3 \in \mathcal{P}(p)$ such that for all $l'' \in \pi_2$, $\mathsf{def}(l'') \neq \mathsf{def}(l)$. Each π_i may be empty.

Recall that, by Definition 4.5, $\mathsf{def}(l)$ is a singleton if l is an assignment, and is empty otherwise. Thus, in this definition, the condition $\mathsf{def}(l) \neq \emptyset$ means that l is an assignment, and $\mathsf{def}(l'') \neq \mathsf{def}(l)$ means that either l'' is not an assignment, or is an assignment to another variable than that defined at l.

In the example shown in Figure 4.2, let us consider the finite syntactic path $\langle 1, 2, 3, 4, 5, 3, 6, 7 \rangle$ corresponding to entering once in the loop and taking the then-branch of the conditional. Since

$$\mathsf{quo} \in \mathsf{def}(1),\ \mathsf{quo} \notin \mathsf{def}(2),\ \mathsf{quo} \notin \mathsf{def}(3) \text{ and } \mathsf{quo} \in \mathsf{ref}(4),$$

there is data dependency of statement 4 on statement 1, i.e. $1 \xrightarrow{\mathcal{D}_d} 4$.

4.2.3 Slice Set

The slice set is the set of labels that must be preserved in the slice.

It is constructed from the slicing criterion, using control and data dependences to decide which labels must be added to it. It contains all the labels on which the slicing criterion is directly or indirectly control or data dependent.

To give the definition of the slice set, we need two notations. Let R denote a binary relation over a set X.

- Let n be a natural number. R^n is defined as the binary relation connecting an element y to an element x if there exists a sequence $u_0 = x, u_1, \ldots, u_{n-1}, u_n = y$ such that for any $0 \leq i < n$, $(u_i, u_{i+1}) \in R$. We can define it recursively as follows:

$$R^n = \begin{cases} \{(x,x) | x \in X\} \text{ if } n = 0 \\ \{(x,y) | \exists z, (x,z) \in R^{n-1} \wedge (z,y) \in R\} \text{ otherwise} \end{cases}$$

  The reflexive and transitive closure R^* of R is the smallest reflexive and transitive relation containing R. It is well-known that:

$$R^* = \bigcup_{n \in \mathbb{N}} R^n$$

- The inverse relation R^{-1} of R is the relation connecting y to x if R connects y to x.

$$R^{-1} = \{(x,y) | (y,x) \in R\}$$

Definition 4.9: Slice set

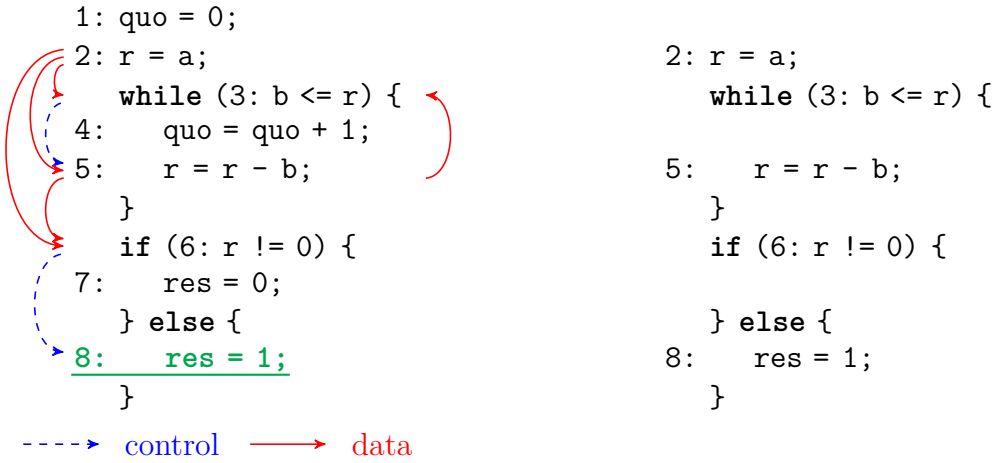
Given a program p and a slicing criterion $C \subseteq L(p)$, the slice set S of p with respect to C is the inverse image of the reflexive and transitive closure of the union of $\mathcal{D}_c$ and $\mathcal{D}_d$. We can note

$$S = \{l \in L(p) \mid \exists l' \in C, l \xrightarrow{(\mathcal{D}_c \cup \mathcal{D}_d)^*} l'\}$$

or

$$S = ((\mathcal{D}_c \cup \mathcal{D}_d)^*)^{-1}(C)$$

Figure 4.4a gives an example of slice set computation, reusing the example presented in Figure 4.2. Figure 4.4a shows the program annotated with the control and data dependencies needed to compute the slice set with respect to $\{8\}$. Line 8 is control dependent on line 6, since it is inside its else-branch. Line 6 reads `r`,



(a) Original program p annotated with dependence information with respect to line 8

(b) Slice q of p with respect to line 8

Figure 4.4 – The original program p and its slice with respect to line 8

hence it is data dependent on lines 2 and 5. Line 5 is control dependent on line 3 and data dependent on line 2 and on itself (not shown in Figure 4.4a since it is not useful for the computation of the slice set). Line 3 is data dependent on line 2. The slice set is thus $\{2, 3, 5, 6, 8\}$.

4.2.4 Quotient

Using control and data dependences, we manipulate only labels, and produce a set of labels, the slice set. But program slicing is expected to produce a program, not a set of statements. The next step is thus to produce a program from the initial program and the slice set.

The goal is to construct a program q from p by removing zero, one or more statements from it, such that the set of labels of q is the slice set. We use a variant of the notion of quotient used in [BBD+10] to formalize the fact that the slice is constructed by removing statements from the initial program (in [BBD+10], parts of the program are replaced by **skip**, while we simply delete them).

Definition 4.10: Quotient

A program p' is said to be a quotient of a program p, denoted $p' \leq_q p$, if this can be deduced from the following set of rules.

$$\lambda \leq_q \lambda \tag{1}$$
$$\text{if } p' \leq_q p,\ \ p' \leq_q s;\ p \tag{2}$$
$$\text{if } s' \leq_q s \text{ and } p' \leq_q p,\ \ s';\ p' \leq_q s;\ p \tag{3}$$
$$\text{if } s \text{ is } \texttt{skip} \text{ or an assignment, } s \leq_q s \tag{4}$$
$$\text{if } p'_1 \leq_q p_1 \text{ and } p'_2 \leq_q p_2,\ \texttt{if}\ (b)\ p'_1\ \texttt{else}\ p'_2 \leq_q \texttt{if}\ (b)\ p_1\ \texttt{else}\ p_2 \tag{5}$$
$$\text{if } p' \leq_q p,\ \texttt{while}\ (b)\ p' \leq_q \texttt{while}\ (b)\ p \tag{6}$$

We describe each rule of Definition 4.10 hereafter.

(1) The empty program is a quotient of itself.

(2) A quotient of a program p is also a quotient of the bigger program $s;\ p$.

(3) The sequence of two quotients is a quotient of the sequence. This rule allows to deeply remove statements from a program, not only the first one as allowed by rule (2).

(4) A simple statement (**skip** or an assignment) is a quotient of itself. This rule allows the quotient relation to be reflexive.

(5) The conditional built using quotients of the branches is a quotient of the conditional. This rule also allows to deeply remove statements in the branches of the conditional.

(6) The loop built using a quotient of the body is a quotient of the loop. Again, this rule allows to deeply remove statements in the body of the loop.

The choice of the symbol "$\leq_q$" to denote the quotient relation is significant. One can show that it is indeed an order relation (i.e. it is reflexive, antisymmetric and transitive).

Given a program $p = s_1;\ s_2;\ s_3$, the programs s_3 and $s_1;\ s_3$ are two quotients of p. Indeed, $s_3 \leq_q p$ by rule (2) applied twice, and $s_1;\ s_3 \leq_q p$ by rules (3) and (2), and by reflexivity of $\leq_q$.

Let us illustrate this notion of quotient using the running example of Figure 4.2. Two possible quotients of it are represented in Figure 4.5a and 4.5b. The first quotient in Figure 4.5a is obtained from p by removing statements 4 and 8. The second quotient in Figure 4.5b is obtained from p by removing every statement

```
1: quo = 0;
2: r = a;                    2: r = a;
   while (3: b <= r) {

5:    r = r - b;                                5:    r = r - b;
    }
   if (6: r != 0) {
7:    res = 0;
    } else {

    }
```

(a) (b) (c)

Figure 4.5 – Examples ((a) and (b)) and counter-example ((c)) of quotients of p (see Figure 4.2)

except 2. The program in Figure 4.5c is not a quotient of p. Indeed, we cannot remove the loop with label 3 without removing the statement of label 5.

Using the notion of quotient, our goal can be rephrased in the following way. Our objective is to construct a quotient of p whose set of labels is the slice set.

With the syntax of our WHILE language, given a program p and a subset of labels $L \subseteq L(p)$, it is clear that there is at most one quotient of p whose set of labels is L. But there can also be no quotient at all. For instance, it is not possible to create the quotient of our program p (cf. Figure 4.2) whose set of labels is $\{5\}$, since, as discussed above, instruction 5 cannot be preserved without instruction 3.

But our slice set is not an arbitrary subset of labels of p. It was built using control and data dependences. We prove that, as soon as control dependence is included in the dependence relation used, a slice set can be associated to a quotient.

Lemma 4.1: Existence of the slice

Let p be a program, C a subset of labels of p and $\mathcal{D}$ a dependence relation on p satisfying $\mathcal{D}_c \subseteq \mathcal{D}$. Then $(\mathcal{D}^*)^{-1}(C)$ is the set of labels of a (uniquely defined) quotient of p.

To prove the existence of such a quotient, we construct it explicitly. Given a set of labels L, we introduce a function named $\mathcal{F}_L$ such that for any program p, $\mathcal{F}_L(p)$ removes from program p all the statements whose labels are not in L. This function does not suppose that L is a slice set, nor even that it contains only labels

in L. Though, it is designed to return the slice with set of labels L when L is a slice set of p.

Definition 4.11

$\mathcal{F}_L$ is defined recursively over $Prog$ as follows:

- For an empty program:
$$\mathcal{F}_L(\lambda) = \lambda$$
- For a sequence of statements:
$$\mathcal{F}_L(s;\ p) = \mathcal{F}_L(s); \mathcal{F}_L(p)$$
- For a **skip** statement or an assignment s with label l:
$$\mathcal{F}_L(s) = \begin{cases} s & \text{if } l \in L \\ \lambda & \text{otherwise} \end{cases}$$
- For an **if** statement:
$$\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) = \begin{cases} \texttt{if}\ (l : b)\ \mathcal{F}_L(q)\ \texttt{else}\ \mathcal{F}_L(r) & \text{if } l \in L \\ \lambda & \text{otherwise} \end{cases}$$
- For **while** statements:
$$\mathcal{F}_L(\texttt{while}\ (l : b)\ q) = \begin{cases} \texttt{while}\ (l : b)\ \mathcal{F}_L(q) & \text{if } l \in L \\ \lambda & \text{otherwise} \end{cases}$$

Given a program p, $\mathcal{F}_L(p)$ inspects each statement of p, tests whether its label is in L, removes it if it not the case and inspects it more deeply otherwise. In particular, in presence of a conditional or a loop, it checks first the label of the statement itself and removes it if the label is not in L, without inspecting the branches or the body at all. This guarantees that any label not in L is removed. On the contrary, this does not guarantee that labels in L are kept. For example, considering program p of Figure 4.2, $\mathcal{F}_{\{5\}}(p) = \lambda$, since every other statement than statement 5 is removed. In particular, the loop with label 3 is removed with its body, which contains statement 5.

Function $\mathcal{F}_L$ has two properties of interest. The first one is that $\mathcal{F}_L(p)$ systematically returns a quotient of p, for arbitrary p. This is Lemma 4.2.

Lemma 4.2

For any program p, $\mathcal{F}_L(p)$ is a quotient of p.

Proof.[2] This is proved by induction on the program p. We detail only the case of the conditional statement. The complete proof is available in the Coq development [Léc16].

Given two programs q and r, let us assume that $\mathcal{F}_L(q)$ and $\mathcal{F}_L(r)$ are quotients of q and r respectively. Let l be a label and b a Boolean expression. Let us show that $\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)$ is a quotient of $\texttt{if}\ (l : b)\ q\ \texttt{else}\ r$. Either $l \in L$ or $l \notin L$.

- Assume $l \in L$. By Definition 4.11,

  $$\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) = \texttt{if}\ (l : b)\ \mathcal{F}_L(q)\ \texttt{else}\ \mathcal{F}_L(r)$$

  By Definition 4.10, since $\mathcal{F}_L(q)$ and $\mathcal{F}_L(r)$ are quotients of q and r respectively, $\texttt{if}\ (l : b)\ \mathcal{F}_L(q)\ \texttt{else}\ \mathcal{F}_L(r)$ is a quotient of $\texttt{if}\ (l : b)\ q\ \texttt{else}\ r$.

- Assume $l \notin L$. By Definition 4.11,

  $$\mathcal{F}_L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) = \lambda$$

  λ is a quotient of every program. In particular, it is a quotient of $\texttt{if}\ (l : b)\ q\ \texttt{else}\ r$.

□

The second property of interest describes the set of labels of $\mathcal{F}_L(p)$, but only in the case where L is a slice set of p for a dependence relation $\mathcal{D}$ that contains $\mathcal{D}_c$. Actually, due to the recursive definition of $\mathcal{F}$, we need to reason inductively about the programs considered. But we cannot allow arbitrary programs, since the link between L and p is important. We introduce the notion of sub-programs, also taken from [BBD+10]. Like quotients, sub-programs are built using pieces of the original program. But while quotients preserve the structure of the original program, creating holes in it, sub-programs are instead continuous portions of it. Reasoning about the sub-programs of p allows to reason inductively in a restrictive setting where we consider only pieces of program p. Since each program manipulated during the inductive proof is a sub-program of p, we are able to make the link between its structure and L. In particular, the interesting property is that the control dependencies of a sub-program of p are included in those of p.

[2] The mechanized version of this proof is available in [Léc16].

Note that we could define the quotients of a given program using the notion of sub-programs, as the programs obtained by removing zero, one or more sub-programs from that program. Actually, this is nearly the definition given in [BBD+10] (but as noted above, in [BBD+10] sub-programs are replaced by **skip**, not removed like we propose here).

Definition 4.12 defines sub-programs formally.

Definition 4.12: Sub-program

A program p' is said to be a sub-program of a program p, denoted $p' \leq_s p$, if this can be deduced from the following set of rules.

$$p \leq_s p \tag{1}$$
$$\text{if } p' \leq_s s \text{ then } p' \leq_s s;\ p \tag{2}$$
$$\text{if } p' \leq_s p \text{ then } p' \leq_s s;\ p \tag{3}$$
$$\text{if } p'_1 \leq_s p_1 \text{ then } p'_1 \leq_s \texttt{if}\ (b)\ p_1\ \texttt{else}\ p_2 \tag{4}$$
$$\text{if } p'_2 \leq_s p_2 \text{ then } p'_2 \leq_s \texttt{if}\ (b)\ p_1\ \texttt{else}\ p_2 \tag{5}$$
$$\text{if } p' \leq_s p \text{ then } p' \leq_s \texttt{while}\ (b)\ p \tag{6}$$

We describe each rule of Definition 4.12 hereafter.

(1) p is a sub-program of itself.

(2) A sub-program of a statement s is a sub-program of the sequence $s;\ p$. Note that this rule deliberately relates a program and a statement, instead of two statements.

(3) A sub-program of a program p is a sub-program of the sequence $s;\ p$.

(4) A sub-program of the first branch of a conditional is a sub-program of the conditional.

(5) A sub-program of the second branch of a conditional is a sub-program of the conditional.

(6) A sub-program of the body of a loop is a sub-program of the loop.

Note that this relation is not symmetrical in the case of the sequence. Given a program $p = s_1;\ s_2;\ s_3$, the program $s_2;\ s_3$ is a sub-program of p thanks to rules (3) and (1), while the program $s_1;\ s_2$ is not a sub-program of p in general.

Like for quotients, the choice of the symbol "$\leq_s$" to denote the sub-program relation is significant, since we can again show that being a sub-program is an order relation.

Given a program p, p is a trival sub-program of itself. Like for quotients, the empty program is a sub-program of every program.

We can illustrate this notion of sub-program more concretely using the running example p of Figure 4.2 and the programs of Figure 4.5. The program in Figure 4.5a is not a sub-program of p since, as noted above, we cannot preserve the first two statements if we do not preserve the whole program. The program in Figure 4.5b is a valid sub-program of p, showing that a program can be both a quotient and a sub-program of p. Figure 4.5c is also a correct sub-program of p. Preserving only the whole loop or the whole conditional would also produce a valid sub-program of p.

We can now state and prove the second property of $\mathcal{F}_L$.

Lemma 4.3

Let p be a program, C a subset of labels of p and $\mathcal{D}$ a dependence relation on p satisfying $\mathcal{D}_c \subseteq \mathcal{D}$. Let S denote the slice set $(\mathcal{D}^*)^{-1}(C)$. Then, for any sub-program q of p, the set of labels of $\mathcal{F}_S(q)$ is $S \cap L(q)$.

Proof.[3] This is proved by induction on the sub-programs of p. We detail only the case of the conditional statement. The complete proof is available in the Coq development [Léc16].

Let $\texttt{if}\ (l : b)\ q\ \texttt{else}\ r$ be a sub-program of p. Let us assume that

$$L(\mathcal{F}_S(q)) = L(q) \cap S$$

and

$$L(\mathcal{F}_S(r)) = L(r) \cap S$$

And let us show that

$$L(\mathcal{F}_S(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)) = L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) \cap S$$

By definition of L,

$$\begin{aligned} L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) \cap S &= (\{l\} \cup L(q) \cup L(r)) \cap S \\ &= (\{l\} \cap S) \cup (L(q) \cap S) \cup (L(r) \cap S) \end{aligned}$$

Either $l \notin S$ or $l \in S$.

[3] The mechanized version of this proof is available in [Léc16].

- Assume that $l \notin S$. Then, by Definition 4.11,

$$L(\mathcal{F}_S(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) = L(\lambda) = \emptyset$$

Since $l \notin S$, thanks to the hypothesis on control dependence and the fact that $\texttt{if}\ (l : b)\ q\ \texttt{else}\ r$ is a sub-program of p, we can deduce that $L(q) \cap S = \emptyset$ and $L(r) \cap S = \emptyset$, otherwise l would have been preserved in S by control dependence. Therefore,

$$(\{l\} \cap S) \cup (L(q) \cap S) \cup (L(r) \cap S) = \emptyset \cup \emptyset \cup \emptyset = \emptyset$$

Thus, for the case $l \notin S$, we can conclude that

$$L(\mathcal{F}_S(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)) = \emptyset = L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) \cap S$$

- Assume now that $l \in S$. In this case, by Definition 4.11,

$$\mathcal{F}_S(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) = \texttt{if}\ (l : b)\ \mathcal{F}_S(q)\ \texttt{else}\ \mathcal{F}_S(r)$$

Thus, we have

$$L(\mathcal{F}_S(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)) = \{l\} \cup L(\mathcal{F}_S(q)) \cup L(\mathcal{F}_S(r))$$

By hypothesis, $L(\mathcal{F}_S(q)) = L(q) \cap S$ and $L(\mathcal{F}_S(r)) = L(r) \cap S$. And since $l \in S$, we have

$$L(\mathcal{F}_S(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)) = (\{l\} \cap S) \cup (L(q) \cap S) \cup (L(r) \cap S)$$

This gives $L(\mathcal{F}_S(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r)) = L(\texttt{if}\ (l : b)\ q\ \texttt{else}\ r) \cap S$ for the case $l \in S$.

□

Based on Lemma 4.2 and Lemma 4.3, we can now prove Lemma 4.1.

Proof of Lemma 4.1.[4] By Lemma 4.2, $\mathcal{F}_S(p)$ is a quotient of p. Since p is a sub-program of itself, it follows from Lemma 4.3 that $L(\mathcal{F}_S(p)) = L(p) \cap S = S$, as S is a subset of the labels of $L(p)$.

Therefore $\mathcal{F}_S(p)$ is a quotient of p containing exactly the labels of $S = (\mathcal{D}^*)^{-1}(C)$.

□

[4]The mechanized version of this proof is available in [Léc16].

4.2.5 Static Backward Slicing

Using the results of the previous section, we can define the program slice as the quotient of the original program whose set of labels is the slice set.

> **Definition 4.13: Slice**
>
> Let p be a program and $C \subseteq L(p)$. The slice of p with respect to C is the quotient of p whose set of labels is $((\mathcal{D}_c \cup \mathcal{D}_d)^*)^{-1}(C)$.

Figure 4.4b represents the program slice of p with respect to $\{8\}$. Only statements on which the slicing criterion depends were preserved (lines 2, 3, 5, 6, 8). The other lines (1, 4, 7) were removed.

4.3 Soundness Property of Program Slicing

The intuition behind static backward slicing is that the slice has the same behavior as the original program with respect to the slicing criterion. In this section, we formalize it using our trajectory-based semantics.

Initially, in Weiser's work [Wei84], the preservation of behavior established by the soundness theorem focused only on the slicing criterion. Informally, the slice is proved to have the same behavior as the original program at the statement in the slicing criterion for the variables in the slicing criterion. Then, Reps et al. [RY89] established that the equivalence of behavior was valid for all statements preserved in the slice, not just at the slicing criterion. Since this second version is strictly stronger than Weiser's one, we present it here.

We first show on a few examples that we cannot compare the trajectory of the initial program and that of the slice directly, and introduce the well-known notion of projection [Wei84] to compare them in Section 4.3.1. Then, we state the soundness theorem of slicing in Section 4.3.2.

4.3.1 Projections

Let us consider the program p of Figure 4.2 and its slice q of Figure 4.4b to give some intuition about the difficulties of the comparison of the trajectories of the initial program and of its slice. Recall that Figure 4.3 presents the trajectory of p on the initial state $\sigma = \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}$. Figure 4.6 represents the trajectory of slice q on the same initial state σ.

First, we can note that the trajectory produced by q contains fewer elements than the one produced by p. This is the direct consequence of q containing fewer statements than p. A first step could be to remove from the trajectory of p each

$$
\begin{aligned}
\mathcal{T}[\![q]\!]\sigma = \langle & (2, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 2, \mathtt{res} \mapsto 0\}) \\
& (3, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 2, \mathtt{res} \mapsto 0\}) \\
& (5, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 1, \mathtt{res} \mapsto 0\}) \\
& (3, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 1, \mathtt{res} \mapsto 0\}) \\
& (5, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}) \\
& (3, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}) \\
& (6, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}) \\
& (8, \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 1\}) \\
\rangle &
\end{aligned}
$$

where

$$\sigma = \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{quo} \mapsto 0, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}$$

Figure 4.6 – Example trajectory of program q of Figure 4.4b

element whose label component was not preserved in q, thus giving traces of equal lengths.

But this is not enough. Even if we only compare the elements with the same labels, the trajectories disagree on the value of `quo`. Indeed, `quo` is modified in p while it totally disappeared from q. This is because all the statements containing it were removed when constructing q. The value of `quo` therefore always remains equal to its initial value during the execution of q, disagreeing with the execution of p as soon as `quo` is modified. We could choose to ignore in the trajectory of p the variables not present in q. It happens that this is enough in this example, but not in general.

Consider the program p' in Figure 4.7a. This program is contrived and is designed only to illustrate our current point. This program assigns variable `a` (line 1) and stores its value in `b` (line 2). Then it gives a value to `c` (line 3), changes the value of `a` (line 4) and stores in `d` the sum of `a` and `c` (line 5). The program q' in Figure 4.7b is the slice of this program with respect to line 5. Statements on lines 3 and 4 were preserved due to data dependencies, statements on lines 1 and 2 were removed.

Consider the initial state

$$\sigma = \{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 0, \mathtt{c} \mapsto 0, \mathtt{d} \mapsto 0\}$$

Figure 4.8 shows the trajectories of p' and q' on σ.

```
1: a = 1;
2: b = a;
3: c = 2;
4: a = 3;
5: d = a + c;
```

(a) Original program p'

```
3: c = 2;
4: a = 3;
5: d = a + c;
```

(b) Program slice q with respect to line 5

Figure 4.7 – A program and its slice with respect to line 5

$$\mathcal{T}[\![p']\!]\sigma = \langle (1, \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 0, \mathtt{c} \mapsto 0, \mathtt{d} \mapsto 0\}) \; (2, \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 0, \mathtt{d} \mapsto 0\}) \; (3, \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 2, \mathtt{d} \mapsto 0\}) \; (4, \{\mathtt{a} \mapsto 3, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 2, \mathtt{d} \mapsto 0\}) \; (5, \{\mathtt{a} \mapsto 3, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 2, \mathtt{d} \mapsto 5\}) \rangle$$

(a) Trajectory of program p' of Figure 4.7a

$$\mathcal{T}[\![q']\!]\sigma = \langle (3, \{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 0, \mathtt{c} \mapsto 2, \mathtt{d} \mapsto 0\}) \; (4, \{\mathtt{a} \mapsto 3, \mathtt{b} \mapsto 0, \mathtt{c} \mapsto 2, \mathtt{d} \mapsto 0\}) \; (5, \{\mathtt{a} \mapsto 3, \mathtt{b} \mapsto 0, \mathtt{c} \mapsto 2, \mathtt{d} \mapsto 5\}) \rangle$$

(b) Trajectory of program q' of Figure 4.7b

where

$$\sigma = \{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 0, \mathtt{c} \mapsto 0, \mathtt{d} \mapsto 0\}$$

Figure 4.8 – Example trajectories of programs of Figure 4.7

Let us ignore the first two elements of the trajectory of p', since they correspond to statements not preserved in q'. Let us look at the third element. Both trajectories agree on the values of `c` and `d`, but disagree on the values of `a` and `b`. Variable `b` does not appear in q', thus we could choose to ignore it in the same way as variable `quo` in our running example. But variable `a` is present in q', on line 4. This shows that only ignoring removed statements and removed variables when comparing the trajectories of the original program and its slice is not enough.

What happens with variable `a` is that it is set on line 1 but never used with this value in the slice. Indeed, when variable `a` is read on line 5, its value comes from the assignment on line 3. When comparing the trajectories, we should focus on the variables whose current values will be used later in the execution of the slice, i.e. on the variables that will be read in the slice before being reassigned. Such variables are named relevant variables in the literature [Wei84, RAB+07]. This is not surprising that we should focus on relevant variables. Indeed, they are the ones captured by data dependence, since data dependence links a statement referencing a variable to the possible last assignments to this variable. If there exists an assignment to this variable that is masked by another assignment in the original program, it is ignored by data dependence and is not necessarily preserved in the slice. When it is not preserved, the behavior of the program slice and the behavior of the original program temporarily disagree on this variable until it is reassigned in a statement in the slice.

Summarizing the last paragraphs, when comparing trajectories, we should remove the elements corresponding to non-preserved statements and, for the remaining ones, we should compare only the values of the variables that are relevant in the slice. Actually, for the second point, for the sake of simplicity, we choose to use an under-approximation of the set of relevant variables. It consists, when comparing two elements of trajectories corresponding to the same statement, in focusing on variables read (or referenced) at that statement. Indeed, it is clear that such variables are relevant variables in the slice, since they are immediately read in that statement. Moreover, since trajectories manipulate states after the execution of statements, we can also include in the comparison the value of the potential variable assigned in the statement. Indeed, if all the variables referenced in the statement have the same values in the behaviors of the original program and the slice, the variable assigned will be given the same value in both executions. This means that we can focus on the variables used at that statement. This is formalized by the following notion of *projection*.

The projection of a state to a set of variables hides the values of variables not in this set.

Definition 4.14: Projection of a state

The projection of a state $\sigma \in \Sigma$ to a set of variables V, denoted $\sigma{\downarrow}V$, is the restriction of σ to V.

Let us illustrate this definition. Given the variables x, y and z and the state $\sigma = \{x \mapsto 0, y \mapsto 1\}$,

- $\sigma{\downarrow}\{x\} = \{x \mapsto 0\}$;
- $\sigma{\downarrow}\{x, y\} = \{x \mapsto 0, y \mapsto 1\}$;
- $\sigma{\downarrow}\{x, z\} = \{x \mapsto 0\}$.

Projecting a trajectory to a set of labels L consists in removing elements whose labels are not in L, and, for each remaining pair (l, σ), projecting the state σ on the set of variables used at statement l, i.e. $\mathsf{used}(l)$.

Definition 4.15: Projection of a trajectory

The projection of a one-element sequence $\langle(l, \sigma)\rangle$ to a set of labels L, denoted $\langle(l, \sigma)\rangle{\downarrow}L$, is defined as follows:

$$\langle(l, \sigma)\rangle{\downarrow}L = \begin{cases} \langle(l, \sigma{\downarrow}\mathsf{used}(l))\rangle & \text{if } l \in L, \\ \langle\,\rangle & \text{otherwise.} \end{cases}$$

The projection of a trajectory $T = \langle(l_1, \sigma_1)\ldots(l_k, \sigma_k)\ldots\rangle$ to L, denoted $\mathsf{Proj}_L(T)$, is defined element-wise:

$$\mathsf{Proj}_L(T) = \langle(l_1, \sigma_1)\rangle{\downarrow}L \oplus \ldots \oplus \langle(l_k, \sigma_k)\rangle{\downarrow}L \oplus \ldots$$

For instance, the projection of the trajectory of Figure 4.3 on the slice set $S = \{2, 3, 5, 6, 8\}$ is presented in Figure 4.9. As discussed above, variable `quo` must be ignored when comparing the trajectories of the initial program and its slice. We can remark that, accordingly, `quo` is absent from the projection.

4.3.2 Soundness Theorem

Using the notion of projection, we can state and prove the soundness property of static backward slicing.

$$\begin{aligned}\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma) = \langle &(2, \{\mathtt{a} \mapsto 2, \mathtt{r} \mapsto 2\}) \\ &(3, \{\mathtt{b} \mapsto 1, \mathtt{r} \mapsto 2\}) \\ &(5, \{\mathtt{b} \mapsto 1, \mathtt{r} \mapsto 1\}) \\ &(3, \{\mathtt{b} \mapsto 1, \mathtt{r} \mapsto 1\}) \\ &(5, \{\mathtt{b} \mapsto 1, \mathtt{r} \mapsto 0\}) \\ &(3, \{\mathtt{b} \mapsto 1, \mathtt{r} \mapsto 0\}) \\ &(6, \{\mathtt{r} \mapsto 0\}) \\ &(8, \{\mathtt{res} \mapsto 1\}) \\ \rangle&\end{aligned}$$

Figure 4.9 – Projection of $\mathcal{T}[\![p]\!]\sigma$ (cf. Figure 4.3) on $S = \{2, 3, 5, 6, 8\}$

Theorem 4.1: Soundness of slicing

Let $C \subseteq L(p)$ be a slicing criterion of program p. Let q be the slice of p with respect to C, and $S = L(q)$ the slice set, i.e. the set of labels preserved in q. Then for any initial state $\sigma \in \Sigma$ of p, if p terminates on σ, then q terminates on σ and:

$$\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma) = \mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma)$$

Proof. [5] Let $\sigma \in \Sigma$. Since p terminates on σ, there exists some $i \geq 0$ such that:

$$\mathcal{T}[\![p]\!]\sigma = \langle (l_1, \sigma_1) \ldots (l_i, \sigma_i) \rangle$$

By Definition 4.15,

$$\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma) = \langle\, (l_{f(1)}, \sigma_{f(1)}{\downarrow}\mathsf{used}(l_{f(1)})) \ldots (l_{f(j)}, \sigma_{f(j)}{\downarrow}\mathsf{used}(l_{f(j)}))\rangle$$

where $j \leq i$ and f is a strictly increasing function.

Moreover, since we cannot assume that q terminates, the trajectory of q on σ is of the form:

$$\mathcal{T}[\![q]\!]\sigma = \langle (l'_1, \sigma'_1)(l'_2, \sigma'_2) \ldots \rangle$$

Because S is exactly the set of labels of q, projecting $\mathcal{T}[\![q]\!]\sigma$ on it does not remove elements from the trajectory, but just projects the states. By Definition 4.15,

$$\mathsf{Proj}_S(\mathcal{T}[\![q]\!]) = \langle\, (l'_1, \sigma'_1{\downarrow}\mathsf{used}(l'_1))(l'_2, \sigma'_2{\downarrow}\mathsf{used}(l'_2)) \ \ldots \rangle$$

[5] The mechanized version of this proof is available in [Léc16].

In particular, $\mathcal{T}[\![q]\!]\sigma$ and $\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma)$ have the same length, i.e. either they are both infinite or they are both finite and have the same length.

In the rest of this proof, we manipulate prefixes of trajectories. We introduce the notation $U^{(k)}$ to denote the prefix of U of size k, for a natural number k and a trajectory U of length at least k (or infinite).

Let us denote by k the greatest natural number such that:

- $(\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma))^{(k)}$ is well-defined, i.e. $k \leq j$;
- $(\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma))^{(k)}$ is well-defined, i.e. $\mathcal{T}[\![q]\!]\sigma$ is infinite or it is finite and k is smaller than or equal to its length;
- $(\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma))^{(k)} = (\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma))^{(k)}$, i.e. the prefixes of length k of the projections of the trajectories of p and q are equal.

We have:

$$\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma))^{(k)} = \langle\, (l_{f(1)}, \sigma_{f(1)}{\downarrow}\mathsf{used}(l_{f(1)})) \ldots (l_{f(k)}, \sigma_{f(k)}{\downarrow}\mathsf{used}(l_{f(k)}))\rangle$$

and

$$\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma))^{(k)} = \langle\, (l'_1, \sigma'_1{\downarrow}\mathsf{used}(l'_1)) \ \ldots \ (l'_k, \sigma'_k{\downarrow}\mathsf{used}(l'_k))\,\rangle$$

Thus, $(\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma))^{(k)} = (\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma))^{(k)}$ means that, for any $m = 1, 2, \ldots, k$, $l_{f(m)} = l'_m$ and $\sigma_{f(m)}{\downarrow}\mathsf{used}(l_{f(m)}) = \sigma'_m{\downarrow}\mathsf{used}(l'_m)$.

Let us prove that $k = j$. We reason by contradiction and assume that $k < j$. By maximality of k, there can be three different cases:

1. $\mathcal{T}[\![q]\!]\sigma$ is of size k, i.e. the trajectory $\mathcal{T}[\![q]\!]\sigma$ ends prematurely, or
2. l'_{k+1} exists, but $l_{f(k+1)} \neq l'_{k+1}$, i.e. the $(k+1)$-th elements of $\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma))$ and $\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma)$ disagree on the label components, or
3. l'_{k+1} exists, $l_{f(k+1)} = l'_{k+1}$, but $\sigma_{f(k+1)}{\downarrow}\mathsf{used}(l_{f(k+1)}) \neq \sigma'_{k+1}{\downarrow}\mathsf{used}(l'_{k+1})$, i.e. the $(k+1)$-th elements of $\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma))$ and $\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma)$ agree on the label components but not on the state components.

Since $l'_k = l_{f(k)}$, the executions of p and q are at the same program point at that execution step. Given our language, if both executions disagree afterwards, either because q terminates (case 1) or because it reaches a program point different from the one reached by p (case 2), this can only be because a control flow statement (i.e. **`if`** or **`while`**), situated in the execution of p between $l_{f(k)}$ and $l_{f(k+1)-1}$, is not present in q or evaluated differently in the executions of p and q.

If such a statement occurred at label $l_{f(k)} = l'_k$, its condition would be evaluated identically in both executions since $\sigma_{f(k)}{\downarrow}\mathsf{used}(l_{f(k)}) = \sigma'_k{\downarrow}\mathsf{used}(l'_k)$. If such a

statement occurred between $l_{f(k)+1}$ and $l_{f(k+1)-1}$ in the execution of p, this would mean that $l_{f(k+1)}$ is part of the body of some non-preserved **if** or **while** statement, which is impossible by definition of control dependence (cf. Definition 4.3).

Thus cases 1 and 2 are impossible. The only possible cause of divergence between the two projections is case 3, i.e. l'_{k+1} exists and is equal to $l_{f(k+1)}$, but the projected states are disjoint ($\sigma_{f(k+1)}{\downarrow}\mathsf{used}(l_{f(k+1)}) \neq \sigma'_{k+1}{\downarrow}\mathsf{used}(l_{f(k+1)})$.

In case 3, the key idea is to remark that

$$\sigma_{f(k+1)-1}{\downarrow}\mathsf{ref}(l_{f(k+1)}) = \sigma'_k{\downarrow}\mathsf{ref}(l_{f(k+1)})$$

i.e. that all the variables read at statement $l_{f(k+1)}$ are evaluated similarly in the states reached by the executions of p and q before the execution of statement $l_{f(k+1)}$, which are $\sigma_{f(k+1)-1}$ and σ'_k respectively.

To prove this, assume that there exists a variable $v \in \mathsf{ref}(l_{f(k+1)})$ such that $\sigma_{f(k+1)-1}(v) \neq \sigma'_k(v)$. Variable v was assigned previously in the execution of p, otherwise it would have the same value in both executions, equal to its initial value in σ. The last assignment to v in the execution of p before its use at $l_{f(k+1)}$ must be preserved in q because of data dependence (cf. Definition 4.8), so it has a label $l_{f(u)} = l'_u$ for some $1 \leq u \leq k$. By definition of k, the state projections after this statement are equal: $\sigma_{f(u)}{\downarrow}\mathsf{used}(l_{f(u)}) = \sigma'_u{\downarrow}\mathsf{used}(l'_u)$, so the last values assigned to v before its use at $l_{f(k+1)}$ are equal, which contradicts the assumption $\sigma_{f(k+1)-1}(v) \neq \sigma'_k(v)$.

This shows that all the variables referenced in $l_{f(k+1)}$ have the same values in both execution, so the resulting states cannot differ, and case 3 is not possible either.

Therefore, neither case 1, case 2, nor case 3 are possible. We can conclude that $k = j$, and we have:

$$\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma) = (\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma))^{(j)}$$

The only case that we need to exclude is the case where the projected execution of q is longer than that of p. This corresponds to the case where $(\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma))^{(j)}$ is a strict prefix of $\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma)$. Assume that this is the case. This means as before that a control flow statement executed in p causes the divergence of the two trajectories. This divergence is due to an **if** or a **while**. By the same reasoning as in cases 1 and 2 above, we show that its condition must be evaluated in the same way in both trajectories and thus it cannot lead to a divergence.

Therefore, $(\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma))^{(j)} = \mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma)$, which gives the desired result:

$$\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma) = \mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma)$$

□

Using our example (cf. Figure 4.4), we can verify that on

$$\sigma = \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{q} \mapsto 0, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0\}$$

the projection of the trajectory of q shown in Figure 4.6 is identical to the projection of the trajectory of p shown in Figure 4.9.

In this chapter, we defined and illustrated classic static backward slicing based on control and data dependencies on a small imperative language. We expressed formally the soundness property that guarantees the equivalence of behaviors of a program and its slice: if the initial program terminates, so does the slice, and both executions agree after each preserved statement on the values of the variables occurring in that statement.

In the next chapter, we use the same concepts and notations to justify the use of slicing for error detection.

Chapter 5

Justification of Program Slicing for Verification on a Representative Language

The previous chapter (Chapter 4) presents classic static backward program slicing on a small WHILE language in an ideal case. Indeed, the soundness theorem established (Theorem 4.1) considers only finite executions of the initial program. Moreover, the trajectory-based semantics used for this language (see Section 4.1.2) gives a meaning to every statement as soon as the initial state considered gives a value to every variable occurring in that statement. This means that statements that can raise runtime errors, such as integer division or array access, are silently excluded from this language.

To apply slicing in the context of verification, we must allow trajectories possibly infinite or with errors. The first step is thus to propose an extension of the language to introduce errors. This is done using assertions that produce errors when failing, i.e. when their Boolean condition is evaluated to false. Instead of adding significantly more dependencies to have a soundness property similar to the classic one (cf. Theorem 4.1), we keep dependencies similar to the classic case and establish a new, weaker soundness property for this language. We call *relaxed slicing* this approach of slicing where we prefer the smallness of the slices to the strength of the soundness theorem. From this new soundness property, we are able to deduce the link between the presence or the absence of errors in the original program and in its slices. This development is formalized in the Coq proof assistant to ensure a high confidence in the results.

This chapter is organized as follows. Section 5.1 presents the new extended language. Next, Section 5.2 introduces a third dependence in addition to the two classic dependence relations and uses them to define program slicing on this WHILE language with errors. Section 5.3 states the new soundness property that

connects the semantics of an initial program and its slices. Section 5.4 shows how to use the results of Section 5.3 in the context of verification. Section 5.5 presents the Coq development formalizing the concepts of this section and highlights some difficulties encountered. Finally, Section 5.6 presents the related work about slicing for verification and concludes this chapter.

As discussed in Section 5.5, the results of this chapter are mechanically proved in Coq. Nevertheless, paper-and-pencil proofs are given for the sake of completeness. The proofs that have a mechanized counterpart in the Coq formalization [Léc16] are marked as such. Note that a paper-and-pencil proof is not necessarily structured in the same way as its Coq counterpart, even though it proves the same statement.

5.1 Presentation of the WHILE Language with Errors

5.1.1 Syntax

We want to add errors to the WHILE language. For that, we introduce some expressions whose evaluation can fail. We call such expressions error-prone or threatening (we also use these adjectives to denote statements containing such expressions). For instance, we add integer division and array access to the language of expressions. For the sake of simplicity, the arrays that we consider in this language are fixed-size arrays, i.e. arrays whose sizes are known at compile time. Like in Section 4.1.1, the exact language of expressions is left abstract, and we assume that a function vars is provided that returns the set of variables occurring in an expression.

To handle errors in a controlled manner, we choose to model them using assertions. Our assertions have the classic semantics: an assertion takes a Boolean condition as argument, stops the program if the condition is evaluated to false, and does nothing otherwise. We consider that, in the programs we consider, all error-prone statements are protected by assertions. These assertions ensure that, if they pass, the evaluation of the statements they protect will not trigger an error. This implies that errors will only be produced by assertions.

Figure 5.1 gives an example of two error-prone statements, each accompanied by a protecting assertion, in a C-like syntax. In Figure 5.1a, the error-prone expression is an integer division: `k/N`. The protecting assertion ensures that the statement is executed only when `N` is not zero. In Figure 5.1b, the error-prone expression is the access to the `k`-th element of an array `a` of known size `N`. The protecting assertion ensures that the statement is executed only if `k` is a valid index, i.e. if $0 \leq \mathtt{k} < \mathtt{N}$.

```
assert (N != 0);
x = k/N;
```

(a) Division by `N`

```
assert (0 <= k < N);
x = a[k];
```

(b) Access to `a` of known size `N`

Figure 5.1 – Examples of statements with their protecting assertions

$$
\begin{array}{lcl}
Prog & ::= & Stmt^* \\
Stmt & ::= & l : \textbf{skip} \mid \\
 & & l : x = e \mid \\
 & & \textbf{if}\ (l : b)\ Prog\ \textbf{else}\ Prog \mid \\
 & & \textbf{while}\ (l : b)\ Prog \\
 & & l : \textbf{assert}\ (b, l')
\end{array}
$$

where

l, l' : label

e : expression

b : Boolean expression

Figure 5.2 – Syntax of the WHILE language

Adding such protecting assertions is also interesting from the point of view of slicing. Indeed, assertions only manipulate the threatening expressions contained in the statements they protect, not the whole statement. This means that assertions can make reference to fewer variables than the protected statement. For example, in Figure 5.1a, the assertion reads only variable `N` while the protected statement reads variables `k` and `N` and defines variable `x`. Choosing as slicing criterion the assertion rather than the protected statement may lead to significantly smaller slices.

This hypothesis that threatening statements are systematically protected by assertions is not too restrictive, since these assertions can be added statically and automatically based on the syntax of the program. This is true in our small language, but even for real languages. For example, in the C language, the RTE plug-in of the Frama-C platform [KKP+15] can add such assertions. This hypothesis is also satisfied in the SANTE method presented in Section 1.3 after the value analysis step.

Like for the definition of the WHILE language without assertions (Definition 4.1), we give to **else** and **while** higher precedences than the sequence operator ";". Again, we use curly brackets ("{" and "}") to disambiguate the programs when

```
l3: assert (z != 0, l1);
l2: assert (w != 0, l);
l1: assert ((y/z) + 1 != 0, l);
l : z = x / ((y/z) + 1) + v/w;
```

(a) Chained assertions

```
l1: assert (k != 0, l);
    while (l : j <=  N/k) {
        ...
l2:     assert (k != 0, l);
    }
```

(b) Protection of a loop condition

Figure 5.3 – Two examples illustrating the use of labels in assertions

needed. Note that this ambiguity is not present in the Coq formalization, since we define the language by its abstract syntax (cf. Definition 5.12).

The syntax of the new WHILE language with errors is given in Figure 5.2. The only difference between this new syntax and the old one (cf. Figure 4.1) is the introduction of the **assert** statement. This **assert** statement is the only statement which contains two labels. The first one is, like in the other statements, the label of the assertion itself. The second one is the label of the protected statement. An assertion often protects the following line. But this second label is particularly useful in two cases (see Figure 5.3).

The first case is when some assertions need to be themselves protected by other assertions because they contain a threatening expression. For instance, in Figure 5.3a, the statement of label `l` contains several threatening sub-expressions. Two assertions of labels `l1` and `l2` are added to protect `l`. But the assertion of label `l1` contains itself a threatening expression. A third assertion of label `l3` is added to protect `l1`. In this example, at least one assertion cannot be located just before the instruction it protects. The label is used to establish the link between the assertions and the right statements.

The second case is when the condition in a **while** contains a threatening expression. In this case, two assertions are added: one before the **while**, and one at the end of the loop body. This is illustrated in Figure 5.3b, in which two assertions of labels `l1` and `l2` are added to protect the loop condition of label `l`. This second assertion of label `l2` is obviously not located before the condition of the **while**. The explicit label is thus needed to indicate that this assertion protects the loop of label `l`.

This syntax of assertions is flexible enough to allow the user to add its own

assertions in a program and check other properties than runtime errors. In this case, assertions are actually used like standard assertions in major programming language (e.g. in C). By inserting custom assertions, the user can insert the checks he wants in the program, e.g. verifying that a variable is positive before executing some piece of code. It can happen in this case that the second label is not needed. But the syntax of the language requires two labels for each assertion. A solution is to fill in this second label with a label not present. Another solution is to fill it in with the label of the assertion itself. In this document, though, we will not introduce any user-defined assertions in the programs we will consider. Thus, we do not go into more detail about them. Note that the user-added assertions are no exceptions to the rule and have to be themselves protected by assertions if necessary.

Figure 5.4 presents a program p written in our new language. This time, line numbers are used for labels. This program takes as input an array `a` of size `N` and an integer `k` such that both $0 \leq \texttt{N} \leq 100$ and $0 \leq \texttt{k} \leq 100$. We also assume that all the values in `a` are bounded between 0 and 100. Moreover, we assume that operations on the integers are done modulo a given maximal integer. All this allows us to ignore overflow problems, i.e. to consider that all arithmetic operations in this program are well-defined and thus do not need to be protected by assertions.

This program computes the average of the values of `a` in two different (and buggy) ways, assuming that it has non-zero values only at indices multiple of `k`, compares the two averages obtained and set `res` to 1 if they are equal and 0 otherwise. The two ways of computing the average of the values of `a` are supposed to be equivalent, thus the expected final value of `res` is 1. This is written as a contract to the program in an ACSL-like syntax [BFH+16]. The line beginning with "requires" specifies the precondition of the program: all values of `a` at indices that are not divisible by `k` are equal to zero. The line beginning with "ensures" specifies the postcondition of the program: `res` is equal to 1 at the end of the program.

In the first part of the program (lines 3 – 8), the index `i` varies from `0` to the largest multiple of `k` less than `N`, and is increased at each iteration by `k`. The sum of the values of `a` at indices i is stored in `s1`. Note that if $\texttt{k} = 0$, the loop on line 4 clearly runs indefinitely (because variable `i` is not modified in the loop body) if the execution enters that loop at least once, i.e. if $0 < \texttt{N}$.

In the second part of the program (lines 9 – 16), the index `j` varies between 0 and `N/k` and is increased by 1 at each iteration. The sum of the values of `a` at indices `k*j` is stored in `s2`. Note that if `k` divides `N`, the last value of `k*j` is equal to `N`, producing an out-of-bounds access to `a`. If `k` does not divide `N`, this second loop computes correctly the sum of the elements of `a`.

requires $\forall i < \mathtt{N}, \neg(\mathtt{k} \mid i) \implies \mathtt{a}[i] = 0$
ensures $\mathtt{res} = 1$

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N, 6);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N, 14);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0, 18);
18 avg1 = s1 / N;
19 assert (N != 0, 20);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   res = 1;
23 else
24   res = 0;
```

Figure 5.4 – Program p computing in two ways the average of the values of an array `a` of size `N`, provided that only values at indices multiple of `k` are not zero

The last part of the program (lines 17–24) computes the averages `avg1` and `avg2` respectively from `sum1` and `sum2` and compares them, setting `res` to 1 is they are equal and to 0 otherwise.

In this program, there are multiple error-prone expressions: some divisions by `N` (lines 11, 18 and 20) and some accesses to `a` (lines 6 and 14). In accordance with our hypotheses, protecting assertions are systematically introduced (lines 5, 10, 13, 17 and 19) to ensure that errors can be triggered only by assertions.

This example program is rather contrived. However, it is a good illustration for our purpose, since it contains both potential errors and infinite loops. Moreover, as it has already been mentioned and as we will see in more detail hereafter, it contains multiple bugs.

5.1.2 Semantics

We define the semantics of the WHILE language with errors in the same way as the semantics of the WHILE language introduced in Section 4.1.2. In particular, the restriction that a program can be executed only on its initial states, i.e. the states giving a value to every variable occurring in that program, still holds. The only difference is the addition of the semantics of assertions.

An assertion stops the execution of the program in an error state if its condition is evaluated to false. It behaves like a **`skip`** statement otherwise. We extend our set of states Σ into $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, where ε denotes an error state, but still use only valid states, i.e. states in Σ, for initial states. We again define a denotational trajectory-based semantics $\mathcal{T}$ of type:

$$\mathcal{T} : Prog \times \Sigma \to Seq(L \times \Sigma_\varepsilon)$$

The type of $\mathcal{T}$ reflects the fact that the initial state must be valid, while errors can appear during the execution.

We reuse the notations $\oplus$, LS_σ and $(v \to T, T')$ introduced in Chapter 4.

As a reminder, $T_1 \oplus T_2$ is the standard concatenation of T_1 and T_2 if T_1 is finite. If T_1 is infinite, then $T_1 \oplus T_2 = T_1$ for any T_2 (and even if T_2 is not well-defined, in other words, $\oplus$ performs lazy evaluation of its arguments). We add another case to the laziness of the $\oplus$ operator. Indeed, we define $T_1 \oplus T_2$ as equal to T_1 if T_1 is infinite as before, but also if it ends with the error state ε. In both cases, T_2 is not evaluated and is allowed to be ill-defined.

LS_σ is defined exactly as in Section 4.1.2. For a finite trajectory T and a state $\sigma \in \Sigma$, we define $LS_\sigma(T)$ as the last state of T (i.e. the state component of its last element) if $T \neq \langle\,\rangle$, and σ otherwise.

The notation $(v \to T, T')$ is also defined in the same way as in Section 4.1.2. Given a Boolean value v and two trajectories T and T', the notation $(v \to T, T')$

is defined as

$$(v \to T, T') = \begin{cases} T \text{ if } v = \mathit{True} \\ T' \text{ if } v = \mathit{False} \end{cases}$$

The semantics of expressions is again denoted $\mathcal{E}$. Given an expression e and a valid state $\sigma \in \Sigma$, $\mathcal{E}[\![e]\!]\sigma$ is defined as before as the evaluation of e using σ to give a value to the variables occurring in e. Thanks to our assumption that statements containing error-prone expressions are protected by assertions, we do not have to define $\mathcal{E}[\![e]\!]\varepsilon$. Indeed, in the same way as we execute programs only on valid states, we evaluate expressions only in valid states.

$\mathcal{T}$ is again defined by recursion over the language. This definition is given hereafter.

Definition 5.1

The trajectory $\mathcal{T}[\![p]\!]\sigma$ of a program p on initial state σ is recursively defined as follows:

$$\begin{aligned}
\mathcal{T}[\![\lambda]\!]\sigma &= \langle\,\rangle && (1)\\
\mathcal{T}[\![s;\ p]\!]\sigma &= \mathcal{T}[\![s]\!]\sigma \oplus \mathcal{T}[\![p]\!](LS_\sigma(\mathcal{T}[\![s]\!]\sigma)) && (2)\\
\mathcal{T}[\![l : \texttt{skip}]\!]\sigma &= \langle (l, \sigma)\rangle && (3)\\
\mathcal{T}[\![l : x = e]\!]\sigma &= \langle (l, \sigma[x \leftarrow \mathcal{E}[\![e]\!]\sigma])\rangle && (4)\\
\mathcal{T}[\![\texttt{if}\ (l : b)\ p\ \texttt{else}\ q]\!]\sigma &= \langle (l, \sigma)\rangle \oplus (\mathcal{E}[\![b]\!]\sigma \to \mathcal{T}[\![p]\!]\sigma, \mathcal{T}[\![q]\!]\sigma) && (5)\\
\mathcal{T}[\![\texttt{while}\ (l : b)\ p]\!]\sigma &= \langle (l, \sigma)\rangle \oplus (\mathcal{E}[\![b]\!]\sigma \to && (6)\\
&\qquad \mathcal{T}[\![p]\!]\sigma \oplus \mathcal{T}[\![\texttt{while}\ (l : b)\ p]\!](LS_\sigma(\mathcal{T}[\![p]\!]\sigma)),\\
&\qquad \langle\,\rangle)\\
\mathcal{T}[\![l : \texttt{assert}(b, l')]\!]\sigma &= (\mathcal{E}[\![b]\!]\sigma \to \langle (l, \sigma)\rangle, \langle (l, \varepsilon)\rangle) && (7)
\end{aligned}$$

Definition 5.1 is questionable in the same way as Definition 4.2. The same idea justifies its existence: the k-th element of the trajectory, when it exists, can be computed in finite time, for any natural k.

Cases (1)–(6) are identical to the ones described in Section 4.1.2. We do not present them again here.

The new case (7) defines the semantics of assertions. Independently of the evaluation of the Boolean expression b, the trajectory of an assertion contains a single element whose label component is equal to the label of the assertion. If b is evaluated to true, the assertion does not modify the input state. If it is evaluated to false, the new state is the error state ε. It should be noted that this last case

is the only case in the definition of $\mathcal{T}$ that is able to produce the error state. Moreover, since $\oplus$ ignores the second operand when the first one ends with ε, if an error state occurs in the trajectory of a program on a given state, it necessarily occurs at the end of the trajectory.

Using our example shown in Figure 5.4, we can illustrate the three possible kinds of trajectory: finite ending without error (see Figure 5.5), finite ending with an error (see Figure 5.6) and infinite (see Figure 5.7). Note that in all three figures, the program is executed on a valid initial state, i.e. a state associating a value to each variable present in the program.

Figure 5.5 illustrates a normal execution of program p of Figure 5.4. In the trajectory, we make abundant use of the notation $\sigma[x \leftarrow v]$, which is the state σ updated with the binding $x \mapsto v$, and its n-ary version

$$\sigma[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n]$$

where $x_1, \ldots, x_n$ are pairwise distinct variables, that updates σ with n bindings. This allows to avoid repeating the values of the variables that are never modified in p.

In Figure 5.5, p is called on a input state where `k` is not equal to 0 and does not divide `N`, thus there is no error during the execution and the averages computed are equal, whence `res` is set to 1 at the end of the execution. More precisely, the execution enters the first loop since `i` $= 0$ and `N` $= 1$. After one iteration, `i` $= 2$, so the execution does not enter the loop again. `last` $= 0$ and $j = 0$, thus the second loop is executed once. Then, the averages `avg1` and `avg2` are computed and compared. Since they are equal, `res` is set to 1.

Figure 5.6 illustrates an execution of program p of Figure 5.4 that ends with an error. Initially, both `N` and `k` are equal to 0. The execution does not enter the first loop. Then the assertion on line 10 fails, and the execution stops here in an error state.

Figure 5.7 illustrates an infinite execution of program p of Figure 5.4. As discussed in Section 5.1.1, since in the initial state `k` is equal to 0 and `N` is different from 0, the loop on line 4 runs indefinitely.

5.2 Dependence-Based Program Slicing on the WHILE Language with Assertions

To define program slicing on this new language, we try to extend in a straightforward way the definitions of the previous chapter. Moreover, to preserve in the slices the hypothesis that threatening statements are protected by assertions, like in the original program, we add a new dependence relation that makes threatening

$$
\begin{aligned}
&\mathcal{T}[\![p]\!]\sigma = \langle \\
&\quad (1,\sigma),(2,\sigma),(3,\sigma),(4,\sigma),(5,\sigma),(6,\sigma[s1 \leftarrow 2]),(7,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2]), \\
&\quad (4,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2]),(9,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2]),(10,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2]), \\
&\quad (11,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0]),(12,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0]), \\
&\quad (13,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0]), \\
&\quad (14,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0,\mathtt{s2} \leftarrow 2]), \\
&\quad (15,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0,\mathtt{s2} \leftarrow 2,\mathtt{j} \leftarrow 1]), \\
&\quad (12,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0,\mathtt{s2} \leftarrow 2,\mathtt{j} \leftarrow 1]), \\
&\quad (17,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0,\mathtt{s2} \leftarrow 2,\mathtt{j} \leftarrow 1]), \\
&\quad (18,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0,\mathtt{s2} \leftarrow 2,\mathtt{j} \leftarrow 1,\mathtt{avg1} \leftarrow 2]), \\
&\quad (19,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0,\mathtt{s2} \leftarrow 2,\mathtt{j} \leftarrow 1,\mathtt{avg1} \leftarrow 2]), \\
&\quad (20,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0,\mathtt{s2} \leftarrow 2,\mathtt{j} \leftarrow 1,\mathtt{avg1} \leftarrow 2,\mathtt{avg2} \leftarrow 2]), \\
&\quad (21,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0,\mathtt{s2} \leftarrow 2,\mathtt{j} \leftarrow 1,\mathtt{avg1} \leftarrow 2,\mathtt{avg2} \leftarrow 2]), \\
&\quad (22,\sigma[\mathtt{s1} \leftarrow 2,\mathtt{i} \leftarrow 2,\mathtt{last} \leftarrow 0,\mathtt{s2} \leftarrow 2,\mathtt{j} \leftarrow 1, \\
&\qquad\quad \mathtt{avg1} \leftarrow 2,\mathtt{avg2} \leftarrow 2,\mathtt{res} \leftarrow 1]) \\
&\rangle
\end{aligned}
$$

where

$$
\begin{aligned}
\sigma = \{&\mathtt{k} \mapsto 2,\mathtt{N} \mapsto 1,\mathtt{a} \mapsto [2],\mathtt{s1} \mapsto 0,\mathtt{s2} \mapsto 0,\mathtt{i} \mapsto 0,\mathtt{j} \mapsto 0, \\
&\mathtt{last} \mapsto 0,\mathtt{avg1} \mapsto 0,\mathtt{avg2} \mapsto 0,\mathtt{res} \mapsto 0\},
\end{aligned}
$$

Figure 5.5 – Example of a finite trajectory ending without error using program p of Figure 5.4

$$
\mathcal{T}[\![p]\!]\sigma = \langle (1,\sigma),(2,\sigma),(3,\sigma),(4,\sigma),(9,\sigma),(10,\varepsilon)\rangle
$$

where

$$
\begin{aligned}
\sigma = \{&\mathtt{k} \mapsto 0,\mathtt{N} \mapsto 0,\mathtt{a} \mapsto [],\mathtt{s1} \mapsto 0,\mathtt{s2} \mapsto 0,\mathtt{i} \mapsto 0,\mathtt{j} \mapsto 0, \\
&\mathtt{last} \mapsto 0,\mathtt{avg1} \mapsto 0,\mathtt{avg2} \mapsto 0,\mathtt{res} \mapsto 0\}
\end{aligned}
$$

Figure 5.6 – Example of abnormal termination using program p of Figure 5.4

$$
\begin{aligned}
&\mathcal{T}[\![p]\!]\sigma = \langle \\
&\quad (1,\sigma),(2,\sigma),(3,\sigma), \\
&\quad (4,\sigma),(5,\sigma),(6,\sigma),(7,\sigma), \\
&\quad (4,\sigma),(5,\sigma),(6,\sigma),(7,\sigma), \\
&\quad (4,\sigma),(5,\sigma),(6,\sigma),(7,\sigma), \\
&\quad \ldots \\
&\rangle
\end{aligned}
$$

where

$$
\begin{aligned}
\sigma = \{&\mathtt{k} \mapsto 0, \mathtt{N} \mapsto 1, \mathtt{a} \mapsto [0], \mathtt{s1} \mapsto 0, \mathtt{s2} \mapsto 0, \mathtt{i} \mapsto 0, \mathtt{j} \mapsto 0, \\
&\mathtt{last} \mapsto 0, \mathtt{avg1} \mapsto 0, \mathtt{avg2} \mapsto 0, \mathtt{res} \mapsto 0\}
\end{aligned}
$$

Figure 5.7 – Example of infinite trajectory using program p of Figure 5.4

statements depend on their protecting assertions, so that these statements cannot be preserved in the slice without the assertions protecting them.

5.2.1 Control Dependence

The definition of control dependence is identical to the one given in the previous chapter, Definition 4.3. Indeed, we choose not to consider assertions as sources of control dependence, and thus, as before, only conditionals and loops can introduce control dependence. Other works have made different choices (see Section 5.6.3).

Definition 5.2: Control dependence $\mathcal{D}_c$

The control dependencies in p are defined by **`if`** and **`while`** statements in p as follows:

- For any statement **`if`** $(l : b)$ q **`else`** r and $l' \in L(q) \cup L(r)$, we define $l \xrightarrow{\mathcal{D}_c} l'$;
- For any statement **`while`** $(l : b)$ q and $l' \in L(q)$, we define $l \xrightarrow{\mathcal{D}_c} l'$.

Like Definition 4.3, Definition 5.2 defines a control dependence relation that is transitive.

For example, considering the program p in Figure 5.4, the assertion at line 5 is control dependent on the loop on line 4, which can be denoted $4 \xrightarrow{\mathcal{D}_c} 5$.

5.2.2 Data Dependence

Data dependence is again defined using finite syntactic paths.

The extension of finite syntactic paths is straightforward.

Definition 5.3: Finite syntactic paths

The set of finite syntactic paths $\mathcal{P}(p)$ of a program p is inductively defined as follows:

$$
\begin{array}{rcll}
\mathcal{P}([\![\lambda]\!]) & = & \{\lambda\} & (1) \\
\mathcal{P}([\![s;\ p]\!]) & = & \mathcal{P}(s) \oplus \mathcal{P}(p) & (2) \\
\mathcal{P}([\![l : \texttt{skip}]\!]) & = & \{l\} & (3) \\
\mathcal{P}([\![l : x = e]\!]) & = & \{l\} & (4) \\
\mathcal{P}([\![\texttt{if}\ (l : b)\ p\ \texttt{else}\ q]\!]) & = & \{l\} \oplus (\mathcal{P}(p) \cup \mathcal{P}(q)) & (5) \\
\mathcal{P}([\![\texttt{while}\ (l : b)\ p]\!]) & = & (\{l\} \oplus \mathcal{P}(p))^* \oplus \{l\} & (6) \\
\mathcal{P}([\![l : \texttt{assert}\ (b, l')]\!]) & = & \{l\} & (7)
\end{array}
$$

The new rule about assertion (Definition 5.3, (7)) describes the only finite syntactic path associated to an assertion: a syntactic path of length 1 containing the label of the assertion, i.e. the same rule as **skip** (Definition 5.3, (3)).

For example, $\langle 1, 2, 3, 4, 9, 10, 11, 12, 17, 18, 19, 20, 21, 22\rangle$ is one finite syntactic path of p of Figure 5.4, where 10, 17 and 19 are labels of assertions.

We introduce the same classic definitions to designate variables read or written at a given statement.

Definition 5.4: Sets def, ref and used

Let l be a label.

- $\mathsf{def}(l)$ denotes the set of variables defined at l (that is, $\mathsf{def}(l) = \{v\}$ if l is an assignment to variable v, and $\emptyset$ otherwise)
- $\mathsf{ref}(l)$ denotes the set of variables referenced (or read) at l
- $\mathsf{used}(l)$ denotes the set of variables defined or read at l, i.e. $\mathsf{used}(l) = \mathsf{def}(l) \cup \mathsf{ref}(l)$

In particular, if l is the label of the assertion $l : \texttt{assert}(b, l')$, $\mathsf{def}(l) = \emptyset$, $\mathsf{ref}(l)$ contains all the variables occurring in b and $\mathsf{used}(l) = \mathsf{ref}(l)$.

For a concrete example, let us consider the assertion on line 5 of program p of Figure 5.4. We have: $\mathsf{ref}(5) = \{\mathtt{i}, \mathtt{N}\}$, $\mathsf{def}(5) = \emptyset$ and $\mathsf{used}(5) = \mathsf{ref}(5) = \{\mathtt{i}, \mathtt{N}\}$.

The definition of data dependence is stated identically to Definition 4.8, but is based on extended versions of finite syntactic paths and def, ref and used sets.

Definition 5.5: Data dependence $\mathcal{D}_d$

Let l and l' be labels of a program p. We say that there is a data dependency $l \xrightarrow{\mathcal{D}_d} l'$ if $\mathsf{def}(l) \neq \emptyset$ and $\mathsf{def}(l) \subseteq \mathsf{ref}(l')$ and there exists a path $\pi = \pi_1 l \pi_2 l' \pi_3 \in \mathcal{P}(p)$ such that for all $l'' \in \pi_2$, $\mathsf{def}(l'') \neq \mathsf{def}(l)$. Each π_i may be empty.

In program p of Figure 5.4, variable `i` is read at 5 and can be last assigned at 3, therefore 5 is data dependent on 3, i.e. $3 \xrightarrow{\mathcal{D}_d} 5$. Variable `i` can also be last assigned at 7, therefore $7 \xrightarrow{\mathcal{D}_d} 5$.

5.2.3 Assertion Dependence

We introduce a third dependence to make threatening statements depend on their protecting assertions. We call it assertion dependence. It is denoted $\mathcal{D}_a$. It simply connects the two labels occurring in an assertion.

Definition 5.6: Assertion dependence $\mathcal{D}_a$

For every assertion l : **`assert`** (b, l') in p with $l, l' \in L(p)$, we define an assertion dependency $l \xrightarrow{\mathcal{D}_a} l'$.

For instance, in our program p shown in Figure 5.4, we have the following assertion dependencies: $5 \xrightarrow{\mathcal{D}_a} 6$, $10 \xrightarrow{\mathcal{D}_a} 11$, $13 \xrightarrow{\mathcal{D}_a} 14$, $17 \xrightarrow{\mathcal{D}_a} 18$ and $19 \xrightarrow{\mathcal{D}_a} 20$.

5.2.4 Slice Set

The slice set is constructed from the slicing criterion, using the three dependence relations: control, data and assertion dependences.

Definition 5.7: Slice set

Given a program p and a slicing criterion $C \subseteq L(p)$, the slice set S of p with respect to C is the inverse image of the reflexive and transitive closure of the union of $\mathcal{D}_c$, $\mathcal{D}_d$ and $\mathcal{D}_a$. We can note

$$S = \{l \in L(p) \mid \exists l' \in C, l \xrightarrow{(\mathcal{D}_c \cup \mathcal{D}_d \cup \mathcal{D}_a)^*} l'\}$$

or

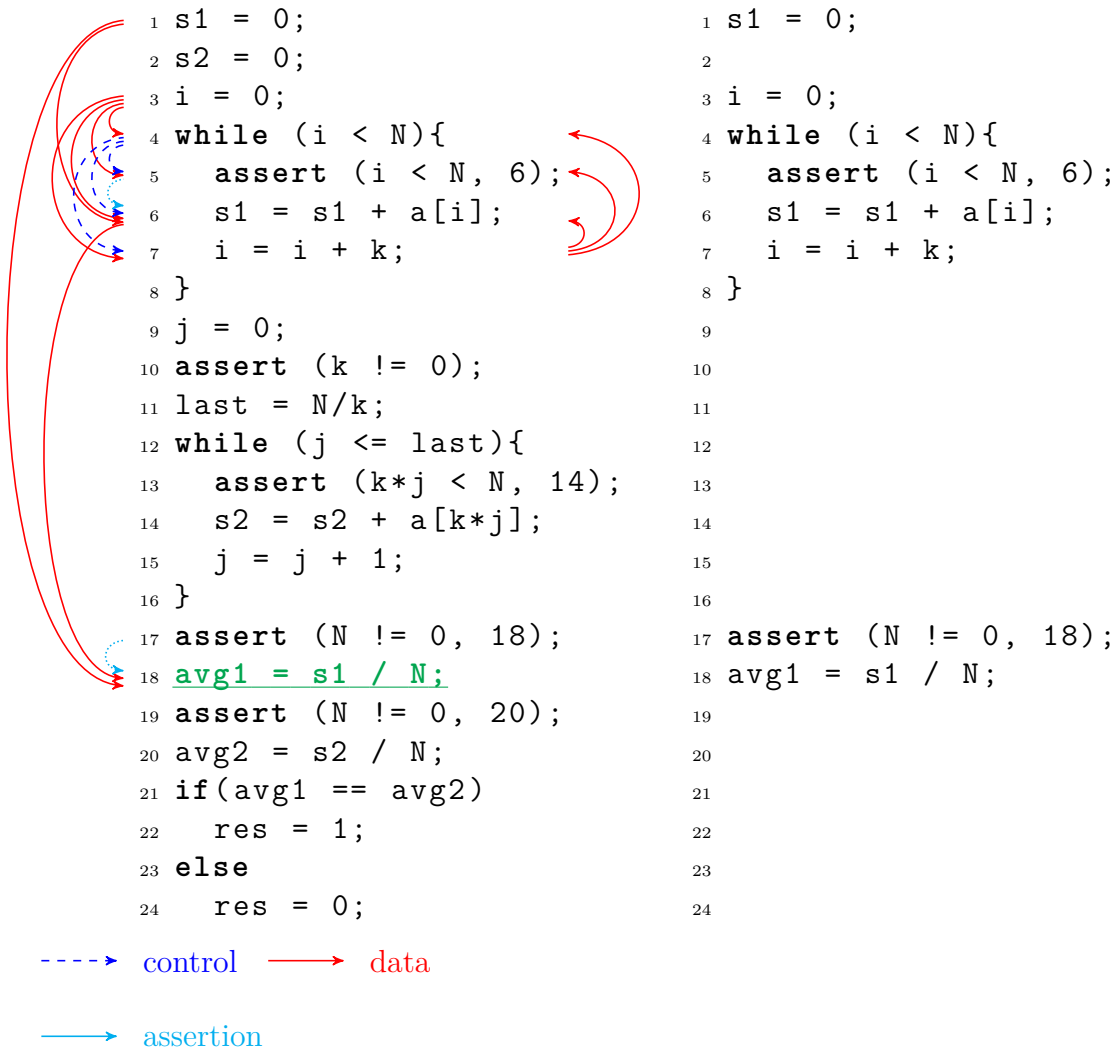
$$S = ((\mathcal{D}_c \cup \mathcal{D}_d \cup \mathcal{D}_a)^*)^{-1}(C)$$

Note that an assertion that is not in the slicing criterion can be added to the slice only using assertion dependence, since it is not the source of any control dependency nor data dependency. Moreover, if a protecting assertion is preserved due to assertion dependence, it will not be responsible of the addition of more statements. Indeed, on the one hand, it is control dependent on the same statements than the protected statement. On the other hand, it manipulates a subset of the variables of the protected statement, it is therefore data dependent on a subset of the statements on which the protected statement is data dependent. Thus, preserving protecting assertions due to assertion dependence does not add more statements to the slice than the protecting assertions themselves.

Figure 5.8a shows the program p of Figure 5.4 annotated with the dependencies needed to compute the slice set with respect to the slicing criterion $\{18\}$ (dependencies of a statement on itself are ignored since they do not change the slice). In addition to control and data dependencies, we have two assertion dependencies: $5 \xrightarrow{\mathcal{D}_a} 6$ and $17 \xrightarrow{\mathcal{D}_a} 18$. The resulting slice set is $S = \{1, 3, 4, 5, 6, 7, 17, 18\}$.

5.2.5 Quotient

Here again, the definition is straightforwardly extended from Definition 4.10.



(a) Original program p annotated with dependence information with respect to line 18

(b) Slice q_1 of p with respect to line 18

Figure 5.8 – The original program p and its slice q_1 with respect to line 18

Definition 5.8: Quotient

A program p' is said to be a quotient of a program p, denoted $p' \leq_q p$, if this can be deduced from the following set of rules.

$$\lambda \leq_q \lambda \tag{1}$$
$$\text{if } p' \leq_q p,\ \ p' \leq_q s;\ p \tag{2}$$
$$\text{if } s' \leq_q s \text{ and } p' \leq_q p,\ \ s';\ p' \leq_q s;\ p \tag{3}$$
$$\text{if } s \text{ is } \texttt{skip}\text{, an assignment or an assertion, } s \leq_q s \tag{4}$$
$$\text{if } p'_1 \leq_q p_1 \text{ and } p'_2 \leq_q p_2,\ \texttt{if}\ (b)\ p'_1\ \texttt{else}\ p'_2 \leq_q \texttt{if}\ (b)\ p_1\ \texttt{else}\ p_2 \tag{5}$$
$$\text{if } p' \leq_q p,\ \texttt{while}\ (b)\ p' \leq_q \texttt{while}\ (b)\ p \tag{6}$$

The assertions are handled the same way as the other simple statements in rule (4).

This definition verifies the same property as before: given a program, there exists one unique quotient whose set of labels is the slice set, provided that control dependencies are included in the considered dependencies.

Lemma 5.1: Existence of the slice

Let p be a program, C a subset of labels of p and $\mathcal{D}$ a dependence relation on p satisfying $\mathcal{D}_c \subseteq \mathcal{D}$. Then $(\mathcal{D}^*)^{-1}(C)$ is the set of labels of a (uniquely defined) quotient of p.

Proof.[1] The proof is really similar to the proof of Lemma 4.1. The presence of assertions just adds one case in the proofs by induction. □

5.2.6 Static Backward Slicing

The program slice is again defined as the quotient of the original program whose set of labels is the slice set.

Definition 5.9: Slice

Let p be a program and $C \subseteq L(p)$. The slice of p based on $\mathcal{D}_c$, $\mathcal{D}_d$ and $\mathcal{D}_a$ with respect to C is the quotient of p whose set of labels is $((\mathcal{D}_c \cup \mathcal{D}_d \cup \mathcal{D}_a)^*)^{-1}(C)$.

Figure 5.8b represents the program slice q_1 of p with respect to $\{18\}$, i.e. the definition of the first average `avg1`. Note that the assertions at line 5 and 17 are preserved thanks to assertion dependence.

[1] The mechanized version of this proof is available in [Léc16].

Since this program has two independent parts, one computing `s1` and `avg1` on the one hand, one computing `s2` and `avg2` on the other hand, slicing is particularly efficient: all statements related to `s2` and `avg2` are absent from the slice.

5.2.7 Illustrating Examples

In this new language allowing errors, and when considering not only finite but also infinite executions, does the same soundness theorem as Theorem 4.1 or some direct extension of it still hold? Previous works on slicing with infinite executions and errors (see Section 5.6) showed in their context that it was not the case.

But let us illustrate some possible cases using our program p of Figure 5.4 and the slices q_1 shown in Figure 5.8b and q_2 shown in Figure 5.9b. We have already introduced q_1 which is the slice of p with respect to the definition of the first average `avg1` on line 18. q_2 is equivalent of q_1 for the second average `avg2`. It is defined as the slice of p with respect to the definition of the second average `avg2` on line 20. Since the computations of `avg1` and `avg2` are completely independent, the two slices q_1 and q_2 have no statement in common.

We provide 5 examples, whose corresponding initial states are named $\sigma_1, \ldots, \sigma_5$. We specify only the inputs `k`, `N` and `a`. Actually, `a` does not influence the behavior of the program nor of its slices, but it is given nevertheless for the sake of clarity. The summary of the results is given in Figure 5.10.

On each execution, each program can have one of the three following behaviors. It can:

- end normally (denoted "—"), or
- end abnormally on a failed assertion at line l (denoted "$\lightning$ line l"), or
- not end at all and run indefinitely due to the loop at line l (denoted "$\circlearrowleft$ line l").

Data σ_1 shows an execution of the original program complying with the specification of the program (cf. the contract in Figure 5.4), while $\sigma_2, \ldots, \sigma_5$ explicit the bugs mentioned above.

- On σ_1, both `s1` and `s2` contain `a[0]` + `a[2]` + `a[4]`, which is the expected value.
- On σ_2, the assertion at line 13 prevents the access of `s2` at an invalid index in line 14. Indeed, if `k` divides `N`, which is the case in σ_2, `k*j` can be equal to `k` $\times$ `last` $=$ `N`, which is not a valid index for `a`. But the error is not so easy to correct, because turning `<=` into `<` in the test `j <= last` would make σ_2 work, but would also make σ_1 compute the wrong sum, namely `a[0]` + `a[2]`.

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N, 6);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N, 14);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0, 18);
18 avg1 = s1 / N;
19 assert (N != 0, 20);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   res = 1;
23 else
24   res = 0;
```

```
1
2  s2 = 0;
3
4
5
6
7
8
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N, 14);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17
18
19 assert (N != 0, 20);
20 avg2 = s2 / N;
21
22
23
24
```

control data

assertion

(a) Original program p annotated with dependence information with respect to line 20

(b) Slice q_2 of p with respect to line 20

Figure 5.9 – The original program p and its slice q_2 with respect to line 20

Initial state		σ_1	σ_2	σ_3	σ_4	σ_5
Inputs		$\texttt{k} = 2$ $\texttt{N} = 5$	$\texttt{k} = 2$ $\texttt{N} = 4$	$\texttt{k} = 0$ $\texttt{N} = 4$	$\texttt{k} = 2$ $\texttt{N} = 0$	$\texttt{k} = 0$ $\texttt{N} = 0$
Example array		$[3, 0, 4, 0, 3]$	$[3, 0, 1, 0]$	$[12, 0, 0, 0]$	$[]$	$[]$
Anomaly	p	—	↯ line 13	↻ line 4	↯ line 13	↯ line 10
	q_1	—	—	↻ line 4	↯ line 17	↯ line 17
	q_2	—	↯ line 13	↯ line 10	↯ line 13	↯ line 10

Figure 5.10 – Errors (↯), non-termination (↻) and normal termination (—) of programs p (of Figure 5.4), q_1 (of Figure 5.8b) and q_2 (of Figure 5.9b) for some inputs.

- σ_3 reveals the infinite loop at line 4 when $\texttt{k} = 0$ and $\texttt{N} \neq 0$.
- σ_4 and σ_5 test the case where the array `a` is empty, i.e. $\texttt{N} = 0$. In σ_4 where $\texttt{k} \neq 0$, this is again a situation where `k` divides `N`. The assertion fails at line 13, like on σ_2.
- In σ_5, $\texttt{k} = 0$, the assertion at line 10 protecting the division by `k` at line 11 fails.

Do the two slices q_1 and q_2 behave like p on these data? On σ_1, the three programs all terminate normally. And on the other test data, when the statement responsible for the abnormal termination or the infinite execution of p is preserved in one of the slice, this slice has the same behavior as p. This is the case of p and q_2 that both fail at line 13 on σ_2, and p and q_1 that both enter an infinite loop on σ_3. On the contrary, when the statement responsible for the termination or the infinite execution of p is not preserved in one of the slices, this slice behaves differently from p. Indeed, the statement being not present in the slice, it cannot influence its behavior. For example, on σ_2, slice q_1 does not contain line 13, thus cannot end on a failed assertion at line 13. Instead, it terminates normally. On σ_3, since q_2 did not preserve the loop at line 4, it cannot enter the same infinite loop as p. Instead, it ends at line 10 where the assertion fails.

Reasoning the other way around, i.e. from the slices to the original program, this shows that if a slice terminates normally, the original program may or may not also end normally. Indeed, it may have encountered earlier a failed assertion or an infinite loop that was not preserved. For instance:

- On σ_1, q_1 ends normally and so does p.

- But on σ_2, q_1 ends normally while p fails at line 13.

Likewise, an error found in a slice may or may not occur in the original program because it may again have encountered earlier a failed assertion or an infinite loop that was not preserved. For instance:

- On σ_2, the error found at line 13 in q_2 can also be found in p.
- On σ_3, q_2 fails at line 10, while p enters an infinite loop at line 4. Because of the infinite loop at line 4, the execution of p never reaches the statement at line 10. We say that the error at line 10 in q_2 is *hidden* by the infinite loop at line 4 in p.
- Likewise, in σ_4, q_1 fails at line 17, while the original program fails at line 13. Because of the failed assertion at line 13, the execution of p never reaches the statement at line 17. We say that the error in q_1 at line 17 is hidden by the error at line 13 in p.

When the error detected by a slice is hidden by an error or an infinite loop in the initial program, this is detected on the particular input tested. When we can find another input where the same error is found in the execution of the initial program, we say that this error is *partially hidden*. This means that, on some inputs, some non-preserved infinite loop or failed assertion hides this error, but on some other inputs, this error is not hidden. For example, on σ_3, the error at line 10 in q_2 is hidden in p by the loop at line 4. However, on σ_5, q_2 still fails on the same error and this time p also fails at line 10. The loop at line 4 therefore hides the error at line 10 partially.

Instead, when an error provoked by a slice cannot be reproduced in the initial program, we say that it is *totally hidden*. This is the case of the error at line 17 in q_1, which is hidden either by the error at line 10, like on σ_5, or the error at line 13, like on σ_4.

We can represent schematically the behaviors observed on the data. This schematic representation is given in Figure 5.11. Due to space constraints, this figure exchanged the rows and the columns of Figure 5.10. Indeed, the rows are the five input states $\sigma_1, \ldots, \sigma_5$, while the columns are the program p and its slices q_1 and q_2.

The cell corresponding to row σ_i and program r contains an abstract representation of r, where statements are represented by lines, and of its behavior on σ_i, represented by a downside arrow along the program representation. The arrow is solid alongside statements, and dashed alongside removed statements (if r is q_1 or q_2). The end of the arrow represents the point of the execution reached. The arrow can end with a symbol representing the kind of termination of the execution:

- If the arrow ends with no symbol, r terminates normally.
- If the arrow ends with $\lightning$, r terminates on a failed assertion.
- If the arrow ends with $\circlearrowleft$, r enters an infinite loop.

Figure 5.11 illustrates graphically the observations made above:

- When a slice does not produce an error, then it does not give any information on the non-preserved statements. For example, a non-preserved statement can produce an error (p and q_1 on σ_2).
- When an error is found in a slice, then either the same error can be found in the original program (p and q_2 on σ_2), or it can be hidden by another error (p and q_1 on σ_4), or it can be hidden by an infinite loop (p and q_2 on σ_3).

These examples clearly show that the equality of projections guaranteed for terminating executions by Theorem 4.1 does not hold in general in this wider context, since the behaviors of the original program and its slice can diverge after an infinite loop or an error in the original program not preserved in the slice. However, as long as the original program has a normal behavior in the non-preserved statements, the comparison between the program and its slice seems reliable. We characterize this formally in the next sections.

5.3 Soundness Property of Relaxed Slicing

The two situations where an error in the slice is hidden by another error or an infinite loop not preserved have in common the fact that this interrupts the projected trajectory of the initial program, while the projected trajectory of the slice can continue growing. For a comparison of the two projections to be meaningful, we must thus compare the projected trajectory of the initial program with the first part of the projected trajectory of the slice, and ignore the second part of the projected trajectory of the slice that represents the behavior of the slice after it had diverged from the behavior of the initial program.

For example, on the input state σ_2 (see Figure 5.10), the execution of program p terminates at line 13, while the execution of q_1 continues and stops (normally) at line 18. We must compare both executions only until line 13.

We formalize this idea using the notion of *prefix*.

5.3.1 Projections

We first need to reintroduce the notions of projections, taking into account the error states. The projection of the error state is itself.

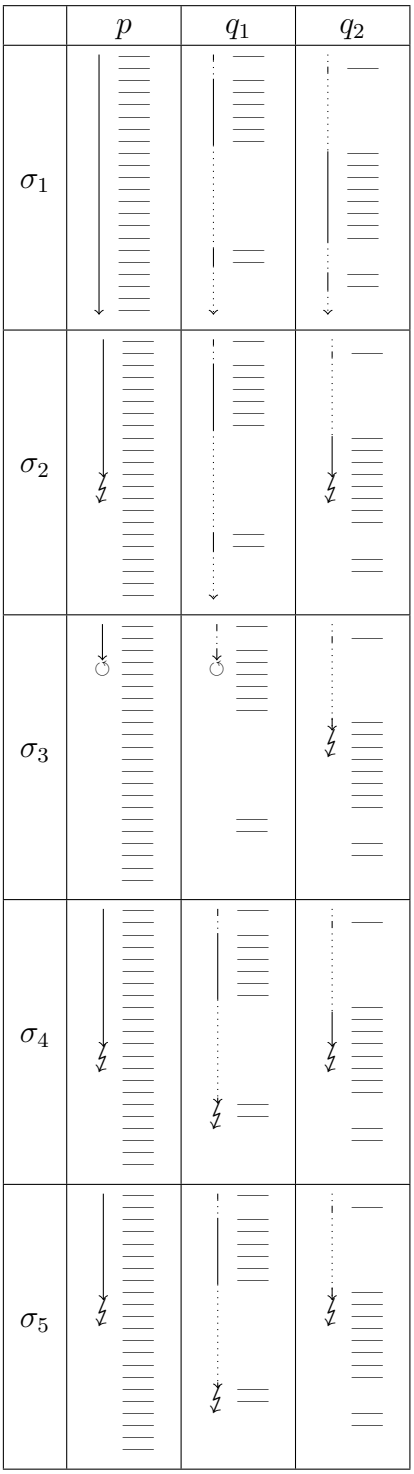


Figure 5.11 – Schematic behaviors of programs p (cf. Figure 5.4), q_1 (cf. Figure 5.8b) and q_2 (cf. Figure 5.9b) for inputs $\sigma_1, \ldots, \sigma_5$.

Definition 5.10: Projection of a state

The projection of a state σ to a set of variables V, denoted $\sigma{\downarrow}V$, is the restriction of σ to V if $\sigma \neq \varepsilon$, and ε otherwise.

The definition of the projection of a trajectory is identical to the classic case (cf. Definition 4.15), with the exception that it uses the projection of states that has just been defined (Definition 5.10).

Definition 5.11: Projection of a trajectory

The projection of a one-element sequence $\langle(l,\sigma)\rangle$ to a set of labels L, denoted $\langle(l,\sigma)\rangle{\downarrow}L$, is defined as follows:

$$\langle(l,\sigma)\rangle{\downarrow}L = \begin{cases} \langle(l,\sigma{\downarrow}\mathsf{used}(l))\rangle & \text{if } l \in L, \\ \langle\,\rangle & \text{otherwise.} \end{cases}$$

The projection of a trajectory $T = \langle(l_1,\sigma_1)\ldots(l_k,\sigma_k)\ldots\rangle$ to L, denoted $\mathsf{Proj}_L(T)$, is defined element-wise:

$$\mathsf{Proj}_L(T) = \langle(l_1,\sigma_1)\rangle{\downarrow}L \oplus \ldots \oplus \langle(l_k,\sigma_k)\rangle{\downarrow}L \oplus \ldots$$

5.3.2 Soundness Theorem

We can now state the soundness property of program slicing based on control, data and assertion dependence relations.

Theorem 5.1: Soundness of slicing

Let $C \subseteq L(p)$ be a slicing criterion of program p. Let q be the slice of p based on control, data and assertion dependences with respect to C, and $S = L(q)$ the set of labels preserved in q. Then for any initial state $\sigma \in \Sigma$ of p and finite prefix T of $\mathcal{T}[\![p]\!]\sigma$, there exists a prefix T' of $\mathcal{T}[\![q]\!]\sigma$, such that:

$$\mathsf{Proj}_S(T) = \mathsf{Proj}_S(T')$$

Moreover, if p terminates without error on σ, $\mathcal{T}[\![p]\!]\sigma$ and $\mathcal{T}[\![q]\!]\sigma$ are finite, and

$$\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma) = \mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma)$$

The proof of Theorem 5.1 is really similar to the proof of the classic soundness theorem (Theorem 4.1). Indeed, if we look closely at the proof of Theorem 4.1, we

can remark that the first part actually proves that the projected trajectory of p is a prefix of the projected trajectory of q, and its second part proves that they are even equal.

The proof of the general case of Theorem 5.1 is thus really similar to the first part of the proof of Theorem 4.1, while the proof of the normal termination case is similar to the second part of the proof of Theorem 4.1. The major difference is the addition of assertions.

Proof.[2] Let $\sigma \in \Sigma$ be an initial state of p,

$$\mathcal{T}[\![p]\!]\sigma = \langle (l_1, \sigma_1)(l_2, \sigma_2) \ldots \rangle$$

and

$$\mathcal{T}[\![q]\!]\sigma = \langle (l'_1, \sigma'_1)(l'_2, \sigma'_2) \ldots \rangle$$

Let $T = \langle (l_1, \sigma_1) \ldots (l_i, \sigma_i) \rangle$ be a finite prefix of $\mathcal{T}[\![p]\!]\sigma$. By Definition 5.11, the projections of T and $\mathcal{T}[\![q]\!]\sigma$ to $S = L(q)$ have the following form

$$\begin{array}{rcl} \mathsf{Proj}_S(T) & = & \langle\, (l_{f(1)}, \sigma_{f(1)}{\downarrow}\mathsf{used}(l_{f(1)})) \ \ldots\ (l_{f(j)}, \sigma_{f(j)}{\downarrow}\mathsf{used}(l_{f(j)}))\,\rangle \\ \mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma) & = & \langle\, (l'_1, \sigma'_1{\downarrow}\mathsf{used}(l'_1))(l'_2, \sigma'_2{\downarrow}\mathsf{used}(l'_2)) \ldots \rangle \end{array}$$

where $j \leq i$ and f is a strictly increasing function. Note that projecting the trajectory of the slice does not remove elements from it, since S is exactly the set of labels of the slice q.

We reintroduce the notation $U^{(k)}$ to denote the prefix of U of size k, for a natural number k and a trajectory U of length at least k (or infinite).

Let us denote by k the greatest natural number such that:

- $(\mathsf{Proj}_S(T))^{(k)}$ is well-defined, i.e. $k \leq j$;
- the prefix of $\mathcal{T}[\![q]\!]\sigma$ of length $(\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma))^{(k)}$ is well-defined, i.e. $\mathcal{T}[\![q]\!]\sigma$ is infinite or it is finite and k is smaller than or equal to its length;
- $(\mathsf{Proj}_S(T))^{(k)} = (\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma))^{(k)}$, i.e. the prefixes of length k of the projections of the trajectories of p and q are equal.

Let $T' = \langle (l'_1, \sigma'_1) \ldots (l'_k, \sigma'_k) \rangle$ be the prefix $(\mathcal{T}[\![q]\!]\sigma)^{(k)}$. By Definition 5.11 and because S is the set of labels of q, we have

$$\mathsf{Proj}_S(T') = (\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma))^{(k)} = \langle\, (l'_1, \sigma'_1{\downarrow}\mathsf{used}(l'_1)) \ \ldots\ (l'_k, \sigma'_k{\downarrow}\mathsf{used}(l'_k))\,\rangle$$

Since, by definition of k, $(\mathsf{Proj}_S(T))^{(k)} = \mathsf{Proj}_S(T')$, we have $l_{f(m)} = l'_m$ and $\sigma_{f(m)}{\downarrow}\mathsf{used}(l_{f(m)}) = \sigma'_m{\downarrow}\mathsf{used}(l'_m)$ for any $m = 1, 2, \ldots, k$. Let us write $\sigma_0 = \sigma'_0 = \sigma$.

Let us prove that $k = j$. We reason by contradiction and assume that $k < j$. By maximality of k, there can be three different cases:

[2] The mechanized version of this proof is available in [Léc16].

1. $\mathcal{T}[\![q]\!]\sigma$ is of size k, i.e. the trajectory $\mathcal{T}[\![q]\!]\sigma$ ends prematurely, or

2. l'_{k+1} exists, but $l_{f(k+1)} \neq l'_{k+1}$, i.e. the $(k+1)$-th elements of $\mathsf{Proj}_S(T)$ and $\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma)$ disagree on the label components, or

3. l'_{k+1} exists, $l_{f(k+1)} = l'_{k+1}$, but $\sigma_{f(k+1)}{\downarrow}\mathsf{used}(l_{f(k+1)}) \neq \sigma'_{k+1}{\downarrow}\mathsf{used}(l'_{k+1})$, i.e. the $(k+1)$-th elements of $\mathsf{Proj}_S(T)$ and $\mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma)$ agree on the label components but not on the state components.

Since $l_{f(k)} = l'_k$, the executions of p and q are at the same program point at that execution step. Given our language, if both executions disagree afterwards, either because q terminates (case 1) or because it reaches a program point different from the one reached by p (case 2), this can only be because a control flow statement (i.e. **`if`**, **`while`** or **`assert`**), situated in the execution of p between $l_{f(k)}$ and $l_{f(k+1)-1}$, is not present in q or is evaluated differently in the executions of p and q.

If such a statement occurred at label $l_{f(k)} = l'_k$, its condition would be evaluated identically in both executions since $\sigma_{f(k)}{\downarrow}\mathsf{used}(l_{f(k)}) = \sigma'_k{\downarrow}\mathsf{used}(l'_k)$. Assume that such a statement occurs between $l_{f(k)+1}$ and $l_{f(k+1)-1}$. This means that it is not preserved in the slice q, since it occurs only in the projected trajectory of p. If this statement is a conditional or a loop, this means that $l_{f(k+1)}$ is part of a branch of this conditional or the body of this loop. Since this statement is not preserved, this is impossible due to control dependence (cf. Definition 5.2). If this statement is an assertion, either the assertion passed and it had no effect on the execution of p, either it failed and stopped the execution of p, which is contradictory since the execution of p reaches $l_{f(k+1)}$.

Thus cases 1 and 2 are impossible. The only possible cause of divergence between the two projections is case 3, i.e. l'_{k+1} exists and is equal to $l_{f(k+1)}$, but the projected states are disjoint ($\sigma_{f(k+1)}{\downarrow}\mathsf{used}(l_{f(k+1)}) \neq \sigma'_{k+1}{\downarrow}\mathsf{used}(l_{f(k+1)})$).

In case 3, the key idea is to remark that

$$\sigma_{f(k+1)-1}{\downarrow}\mathsf{ref}(l_{f(k+1)}) = \sigma'_k{\downarrow}\mathsf{ref}(l_{f(k+1)})$$

i.e. that all the variables read at statement $l_{f(k+1)}$ are evaluated similarly in the states reached by the executions of p and q before the execution of statement $l_{f(k+1)}$, which are $\sigma_{f(k+1)-1}$ and σ'_k respectively.

To prove this, assume that there exists a variable $v \in \mathsf{ref}(l_{f(k+1)})$ such that $\sigma_{f(k+1)-1}(v) \neq \sigma'_k(v)$. v was assigned previously in the execution of p, otherwise it would have the same value in both executions, equal to its initial value in σ. The last assignment to v in the execution of p before its usage at $l_{f(k+1)}$ must be preserved in q because of data dependence (cf. Definition 5.5), so it has a label $l_{f(u)} = l'_u$ for some $1 \leq u \leq k$. By definition of k, the state projections after this statement are equal: $\sigma_{f(u)}{\downarrow}\mathsf{used}(l_{f(u)}) = \sigma'_u{\downarrow}\mathsf{used}(l'_u)$, so the last values

assigned to v before its usage at $l_{f(k+1)}$ are equal, which contradicts the assumption $\sigma_{f(k+1)-1}(v) \neq \sigma'_k(v)$.

This shows that all the variables referenced in $l_{f(k+1)}$ have the same values, so the resulting states cannot differ, and case 3 is not possible either.

Therefore, neither case 1, case 2, nor case 3 are possible. We can conclude that $k = j$, and T' satisfies

$$\mathsf{Proj}_S(T) = \mathsf{Proj}_S(T')$$

If p terminates without error on σ, by the first part of the theorem we have a prefix T' of $\mathcal{T}[\![q]\!]\sigma$ such that $\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma) = \mathsf{Proj}_S(T')$. If T' is a strict prefix of $\mathcal{T}[\![q]\!]\sigma$, this means as before that a control flow statement executed in p causes the divergence of the two trajectories. By assumption, there are no failing assertions in the execution of p, therefore it is due to an **`if`** or a **`while`**. By the same reasoning as in cases 1 and 2 above, we show that its condition must be evaluated in the same way in both trajectories and cannot lead to a divergence. Therefore, $T' = \mathcal{T}[\![q]\!]\sigma$. □

Theorem 5.1 has two parts.

The first part describes the relation between the semantics of the program and its slice in the general case. It is weaker than the classic property (cf. Theorem 4.1). It compares only prefixes of projections of trajectories instead of comparing the whole projections. In formalizing our version of slicing, we chose to produce slices similar to the ones produced by classic slicing, and adapt the soundness theorem, instead of modifying the slices and keep the same soundness theorem. Indeed, this second approach could have produced far larger slices, e.g. if we had decided to keep all statements located after a potential infinite loop or a potential failing assertion. Since we decided to relax the soundness property, we call this version of slicing *relaxed slicing*, and the slices produced *relaxed slices*.

The second part focuses on executions that terminate without error. In this case, the equality of projections of trajectories is guaranteed, like in the classic case (cf. Theorem 4.1). Actually, the soundness property of classic slicing can be seen as a corollary of this second part.

5.4 Verification on Relaxed Slices

In this section, we show how the absence and the presence of errors in relaxed slices can be soundly interpreted in terms of the initial program.

Lemma 5.2

Let q be a relaxed slice of p and $\sigma \in \Sigma$ an initial state of p. If the preserved assertions do not fail in the execution of q on σ, they do not fail in the execution of p on σ either.

Proof.[3] Let us show the contrapositive. Assume that $\mathcal{T}[\![p]\!]\sigma$ ends with (l, ε) where $l \in L(q)$ is a preserved assertion. Let $L = L(q)$. From Theorem 5.1 applied to $T = \mathcal{T}[\![p]\!]\sigma$, it follows that there exists a finite prefix T' of $\mathcal{T}[\![q]\!]\sigma$ such that $\mathsf{Proj}_L(T) = \mathsf{Proj}_L(T')$. The last state of $\mathsf{Proj}_L(T')$ is ε, therefore the last state of T' is ε too. It means that ε appears in $\mathcal{T}[\![q]\!]\sigma$, and by definition of semantics (cf. Section 5.1.2) this is possible only if ε is its last state. Therefore $\mathcal{T}[\![q]\!]\sigma$ ends with (l, ε) as well. □

The following theorem and corollary immediately follow from Lemma 5.2.

Theorem 5.2

Let q be a relaxed slice of p. If all assertions contained in q never fail, then the corresponding assertions in p never fail either.

Corollary 5.1

Let $q_1, \ldots, q_n$ be relaxed slices of p such that each assertion in p is preserved in at least one of the q_i. If no assertion in any q_i fails, then no assertion fails in p.

The last result justifies the detection of errors in a relaxed slice.

Theorem 5.3

Let q be a relaxed slice of p and $\sigma \in \Sigma$ an initial state of p. We assume that $\mathcal{T}[\![q]\!]\sigma$ ends with an error state. Then one of the following cases holds for p:

(i) $\mathcal{T}[\![p]\!]\sigma$ ends with an error at the same label, or

(ii) $\mathcal{T}[\![p]\!]\sigma$ ends with an error at a label not preserved in q, or

(iii) $\mathcal{T}[\![p]\!]\sigma$ is infinite.

[3] The mechanized version of this proof is available in [Léc16].

Proof.[4] Let $L = L(q)$ and assume that $\mathcal{T}[\![q]\!]\sigma$ ends with (l, ε) for some preserved assertion at label $l \in L$. We reason by contradiction and assume that $\mathcal{T}[\![p]\!]\sigma$ does not satisfy any of the three cases. Then two cases are possible.

First, $\mathcal{T}[\![p]\!]\sigma$ ends with (l', ε) for another preserved assertion at label $l' \in L$ (with $l' \neq l$). Then reasoning as in the proof of Lemma 5.2 we show that $\mathcal{T}[\![q]\!]\sigma$ ends with (l', ε) as well, that contradicts $l' \neq l$.

Second, $\mathcal{T}[\![p]\!]\sigma$ is finite without error. Then the second part of Theorem 5.1 can be applied and thus $\mathsf{Proj}_S(\mathcal{T}[\![p]\!]\sigma) = \mathsf{Proj}_S(\mathcal{T}[\![q]\!]\sigma)$. This is contradictory since $\mathcal{T}[\![q]\!]\sigma$ contains an error (at label $l \in L$) and $\mathcal{T}[\![p]\!]\sigma$ does not. □

For instance, consider the example of Figure 5.8 with hypotheses $0 < \mathtt{k} \leq 100$ and $0 < \mathtt{N} \leq 100$. In this case we can prove that slice q_1 does not contain any error, thus we can deduce by Theorem 5.2 that the assertions at lines 5 and 17 (preserved in slice q) never fail in the initial program either. If in addition we replace `N/k` by `(N-1)/k` at line 11 of Figure 5.9, we can show that neither of the two slices of Figure 5.8 and Figure 5.9 contains any error. Since these slices cover all assertions, we can deduce by Corollary 5.1 that the initial program is error-free.

Theorem 5.3 shows that despite the fact that an error detected in q does not necessarily appear in p, the detection of errors on q has a precise interpretation. It can be particularly meaningful for programs supposed to terminate, for which a non-termination within some time τ is seen as an anomaly. In this case, detection of errors in a slice is sound in the sense that if an error is found in q for initial state σ, there is an anomaly (same or earlier error, or non-termination within time τ) in p whose type can be easily determined by running p on σ.

It can be noticed that a result similar to Theorem 5.3 can be established for non-termination: if $\mathcal{T}[\![q]\!]\sigma$ is infinite, then either (ii) or (iii) holds for p.

In the framework of a verification method, Corollary 5.1 and Theorem 5.3 indicate how to safely transpose to the initial program the verification results obtained on the slices. In particular, they properly justify the use of program slicing in SANTE (cf. Section 1.3). More precisely, they show that, in the SANTE method, the interpretation of the errors found by dynamic analysis applied on the slices is correct. In particular, the SANTE method can soundly conclude that a program is safe when all the alarms are classified as false positive when analyzing the slices. The method behind the SYMBIOTIC tool can probably be justified in the same way.

[4] The mechanized version of this proof is available in [Léc16].

```
Inductive prog : Type :=
| P_nil : prog (* the empty program *)
| P_cons : stmt → prog → prog (* a statement plus a program *)
with stmt :=
| S_skip : label → stmt (* skip *)
| S_ass : label → id → aexp → stmt (* assignment *)
| S_if : label → bexp → prog → prog → stmt (* if *)
| S_while : label → bexp → prog → stmt (* while *)
| S_assert : label → bexp → label → stmt. (* assertion *)
```

Figure 5.12 – Language definition

5.5 Remarks about the Coq Formalization

To ensure a high confidence in the results, we formalize the theory of relaxed slicing in the Coq proof assistant. This formalization contains 10 000 lines of Coq code (3 200 lines of specification, 6 500 lines of proof).

We present an overview of the Coq formalization [Léc16] in Section 5.5.1 and focus on some interesting aspects of it in the next subsections.

5.5.1 Overview

The structure of the Coq development [Léc16] follows the formalization of this chapter.

Language The WHILE language defined in Coq is very similar to the one given in Section 5.1.1.

The definition is readable even for those who are not familiar with Coq. It is given in Figure 5.12. The definition is mutually-inductive. `prog` is defined as either the empty program (`P_nil`) or a sequence of a statement and a program (`P_cons`), i.e. `prog` is a list of statements. `stmt` describes the five possible kinds of statements: a **`skip`**, an assignment, a conditional, a loop or an assertion.

In this definition, `id` and `label` are basically `nat`, the type of natural integers, and `aexp` (resp. `bexp`) is the type of arithmetic (respectively Boolean) expressions. For convenience reasons, expressions that can produce an error, such as integer divisions, are not allowed. Expressions of type `aexp` and `bexp` thus trivially cannot produce any error. This is discussed in more detail in Section 5.5.2. The definition of the language and most of the basic notions are strongly inspired by a Coq tutorial [PCG+15].

As it can be seen in the definition, all statements are labelled, and assertions have a second label to point to other statements.

Semantics. The normal states of a program are expressed as functions from identifiers to values (of type `state` := `id` → `nat`). A state is of type

$$\texttt{state_eps} := \texttt{option state}$$

i.e. it is either `None` (the error state) or `Some st`, where `st` is a normal state.

The denotational semantics is given as a recursive function

$$\texttt{traj_prog} : \texttt{nat} \to \texttt{state_eps} \to \texttt{prog} \to \texttt{traj}$$

taking as parameters a desired number of trajectory steps, an initial state and a program, and returning a finite prefix of the full trajectory of the program from this initial state (of type `traj` := `list` (`label` ∗ `state_eps`)).

A similar function

$$\texttt{proj_traj_prog} : \texttt{nat} \to \texttt{state_eps} \to \texttt{prog} \to \texttt{set label} \to \texttt{partial_traj}$$

takes as an additional parameter a set of labels and computes the projected trajectory to this set. Its return type is

$$\texttt{partial_traj} = \texttt{list}\ (\texttt{label} * \texttt{partial_state_eps})$$

where `partial_state_eps` is either the error state or a partially defined state.

Dependences. The dependence relations are defined in a similar way to Section 5.2.

The definition of assertion dependence, shown in Figure 5.13, is the simplest one. As Definition 5.6, it connects the two labels of an assertion. Most of the cases (`ADP_here`, `ADP_after`, `ADS_ifb_l`, `ADS_ifb_r`, `ADS_while`) explore the program recursively, while the key case, really generating dependencies, is `ADS_assert`.

The definition of (unitary) control dependence is given in Figure 5.14 (cf. Definition 5.2). Most of the cases only ensure the recursive descent, as in the definition of `rel_assert`. They are omitted in the figure. Unitary control dependencies can arise from the then-branch (`CDS_ifb_cond_l`) or the else-branch (`CDS_ifb_cond_r`) of a condition, or the body of a loop (`CDS_while_cond`). Relation `rel_top_labels p l` holds if and only if a statement of label `l` is in the top-level sequence of program `p`, i.e. not in a nested condition or loop. This predicate ensures that `rel_control` characterizes only unitary control dependencies. Since slicing manipulates reflexive transitive closures of dependence relations, this will not change the slices.

Data dependence, given in Figure 5.15, is defined in the same spirit as Definition 5.5. It also uses finite syntactic paths, using two predicates `is_flat_of` and `flat_data`. The predicate `is_flat p q` states that `q` is (a specific representation of) a syntactic path of program `p`, while `flat_data q l l'` states that `q` carries the

```
Inductive rel_assert : prog → relation label :=
| ADP_here : forall s p l l', rel_assert_stmt s l l' →
            rel_assert (P_cons s p) l l'
  (* assertion dependence from inside the first statement *)
| ADP_after : forall s p l l', rel_assert p l l' →
             rel_assert (P_cons s p) l l'
  (* assertion dependence from inside
     the remainder of the program *)
with rel_assert_stmt : stmt → relation label :=
| ADS_ifb_l : forall l b p1 p2 l' l'', rel_assert p1 l' l'' →
             rel_assert_stmt (S_if l b p1 p2) l' l''
  (* assertion dependence from inside the then-branch *)
| ADS_ifb_r : forall l b p1 p2 l' l'', rel_assert p2 l' l'' →
             rel_assert_stmt (S_if l b p1 p2) l' l''
  (* assertion dependence from inside the else-branch *)
| ADS_while : forall l b p l' l'', rel_assert p l' l'' →
             rel_assert_stmt (S_while l b p) l' l''
  (* assertion dependence from inside the body *)
| ADS_assert : forall b l l', rel_assert_stmt (S_assert l b l') l l'.
  (* an assertion of label l protecting l'
     gives the dependency l → l' *)
```

Figure 5.13 – Implementation of assertion dependence in Coq (cf. Definition 5.6)

```
Inductive rel_control : prog → relation label :=
| CDP_here : ...
  (* control dependence from inside the first statement *)
| CDP_after : ...
  (* control dependence from inside
     the remainder of the program *)
with rel_control_stmt : stmt → relation label :=
| CDS_ifb_l : ...
  (* control dependence from inside the then-branch *)
| CDS_ifb_r : ...
  (* control dependence from inside the else-branch *)
| CDS_while : ...
  (* control dependence from inside the body *)
| CDS_ifb_cond_l : forall l l' b p1 p2, rel_top_labels p1 l' →
                   rel_control_stmt (S_if l b p1 p2) l l'
  (* control dependencies of (a top-level statement of)
     the then-branch on the condition *)
| CDS_ifb_cond_r : forall l l' b p1 p2, rel_top_labels p2 l' →
                   rel_control_stmt (S_if l b p1 p2) l l'
  (* control dependencies of (a top-level statement of)
     the else-branch on the condition *)
| CDS_while_cond : forall l l' b p, rel_top_labels p l' →
                   rel_control_stmt (S_while l b p) l l'.
  (* control dependencies of (a top-level statement of)
     the body on the condition *)
```

Figure 5.14 – Implementation of control dependence in Coq (cf. Definition 5.2)

```
Inductive rel_data : prog → relation label :=
| DDP_flat : forall p q, is_flat_of q p → forall l l',
             flat_data q l l' → rel_data p l l'.
  (* a data dependency along a path q of p
     is a data dependency in p *)
```

Figure 5.15 – Implementation of data dependence in Coq (cf. Definition 5.5)

data dependency of `l'` on `l`. By collecting data dependencies in every syntactic path of `p`, we reconstruct all the data dependencies of `p`.

Slice computation. As mentioned in Section 5.2, the computation of the slice is divided into two parts: the computation of the set of instructions that must be preserved, called the slice set, and the computation of the slice itself from the original program and the slice set.

For the computation of the slice set, we implement three functions named `control_prog`, `data_prog` and `assert_prog` that compute the control, data and assertion dependencies respectively of a program as a list of pairs of labels, and that are proved correct with respect to `rel_control`, `rel_data` and `rel_assert` respectively.

While `control_prog` and `assert_prog` can be written easily, `data_prog` is more difficult to implement. This is discussed in Section 5.5.6.

`control_prog`, `data_prog` and `assert_prog` are used in a function denoted

$$\texttt{set_slice} : \texttt{prog} \rightarrow \texttt{set label} \rightarrow \texttt{set label}$$

`set_slice p L` computes the inverse image of the slicing criterion `L` under the reflexive transitive closure of the union of the dependence relations.

As for the second part, the computation of the slice from the slice set, it is implemented by a function called `slice : prog → set label → prog`. `slice p L` computes the quotient of `p` whose set of labels is `set_slice p L`. It uses a function

$$\texttt{keep_L_prog} : \texttt{prog} \rightarrow \texttt{set label} \rightarrow \texttt{set label}$$

which is the equivalent in Coq of function $\mathcal{F}_L$ (cf. Definition 4.11), and which basically iterates on the statements of the program and removes those whose labels are not to be preserved.

Results. The main results, presented in Figure 5.16, formalize in Coq the soundness of relaxed slicing (Theorem 5.1). The first part of Theorem 5.1 is implemented by theorem `slice_proj_prefix` in Figure 5.16a. In this theorem, `prefix` formalizes a standard notion of prefix for lists. Thus, the theorem in Figure 5.16a expresses in Coq that the projection of the trajectory of the initial program is a prefix of the projection of the trajectory of its slice.

The second theorem (given in Figure 5.16b) is more complex, since we manipulate finite prefixes instead of the whole trajectory (this is discussed in Section 5.5.4). The condition `length (traj_prog N0 ste p) < N0` ensures that the execution of `p` stopped before `N0` steps.

$$\texttt{last_state (traj_prog N0 ste p) ste <> None}$$

```
Theorem slice_proj_prefix :
  forall p L n ste,
  prefix (proj_traj_prog n ste p (set_slice p L))
         (proj_traj_prog n ste (slice p L) (set_slice p L)).
```

(a) Soundness theorem in the general case

```
Theorem slice_proj_equal :
  forall n ste p L,
  length (traj_prog n ste p) < n →
  (* the execution of p terminates in less than n steps *)
  last_state (traj_prog n ste p) ste <> None →
  (* the execution of p does not reach an error *)
  length (traj_prog n ste (slice p L)) < n ∧
  (* the slice terminates in less then n steps *)
  last_state (traj_prog n ste (slice p L)) ste <> None ∧
  (* the slice does not reach an error *)
  proj_traj_prog n ste p (set_slice p L) =
  proj_traj_prog n ste (slice p L) (set_slice p L).
  (* the projections of the trajectories are equal *)
```

(b) Soundness theorem in the case of normal termination

Figure 5.16 – Soundness theorem (cf. Theorem 5.1) formalized in Coq

```
1: q := 0;
2: r := a;
WHILE 3: b <= r DO
    4: q := q + 1;
    5: r := r - b
END;
IF 6: not (r == 0) THEN
    7: res := 0
ELSE
    8: res := 1
FI
```

Figure 5.17 – Program p of Figure 4.2 in the syntax accepted by the extracted slicer

expresses that the last state is not an error. Together, those conditions characterize a normal termination for the execution of `p` from `ste` in less than `N0` steps.

From these theorems, we can prove the formalizations in Coq of the two theorems of Section 5.4. They can be found in the Coq development [Léc16].

Extraction. The function `slice` can be extracted into OCaml, giving a certified implementation of a slicer for the WHILE language with errors. Adding a wrapper to read a file and adapting the simple parser described in [PCG+15], this gives an executable program reading a program in a file and displaying the slice. Due to limitations in the parser, the names of the variables are lost during parsing, thus new names are generated for the printing. We choose to display both the original program with the new variable names and the slice so that the user can easily compare the two programs.

Figure 5.17 presents the program of Figure 4.2 written in the concrete syntax accepted by the parser.

Calling the extracted slicer on the file containing this program with the additional argument `[8]` produces the slice of the program with respect to the statement of label 8. The exact output is given in Figure 5.18. As expected, statements 1, 4 and 7 are sliced away.

5.5.2 WHILE language with Errors

The WHILE language defined in Coq is slightly less expressive than the one presented in Section 5.1.1. Indeed, for convenience reasons, elements that can lead to threatening expressions such as integer division or arrays are not included in the language. This trivially guarantees the hypothesis that all threatening statements

```
$ ./test_slice.byte divides.prog \[8\]
Original program:
1: x0 := 0;
2: x1 := x2;
WHILE 3: (x3) <= (x1) DO
    4: x0 := (x0) + (1);
    5: x1 := (x1) - (x3)
END;
IFB 6: not ((x1) == (0)) THEN
    7: x12 := 0
ELSE
    8: x12 := 1
FI

Slice:
2: x1 := x2;
WHILE 3: (x3) <= (x1) DO
    5: x1 := (x1) - (x3)
END;
IFB 6: not ((x1) == (0)) THEN
    ''
ELSE
    8: x12 := 1
FI
```

Figure 5.18 – Output of the extracted slicer on the program of Figure 5.17

are protected by assertions, since there are none. This language is still representative for our purpose, though, since it contains errors modeled as assertions and possible infinite loops.

5.5.3 Formalization of States

In this document, we make the assumption that, given a program, we consider as initial states only valid states, i.e. states that differ from the error state and define every variable occurring in the program. This hypothesis is not made in Coq.

First, in Coq, states are formalized using total functions, thus systematically giving a value to every possible variable. In Coq's language, states have type `state` := `id` → `nat`, where `id` is the type of variables (an alias to `nat` actually).

Second, for the sake of simplicity, we allow a pervasive use of error states. For example, the function computing the trajectory of a program, described in more detail hereafter, takes a `state_eps` := `option state` as parameter, where `None` denotes the error state ε and `Some s` denotes a valid state `s`.

5.5.4 No Use of Coinductive Types

Coinductive types in Coq allow to model countably infinite data whose finite prefixes are computable in finite time. For example, Figure 5.19 illustrates how to define coinductive lists in Coq. `infList` is a polymorphic type of finite and infinite lists. Note that the definition of `infList` is identical to the definition of standard inductive lists (cf. Figure 2.1), except `CoInductive` is used in place of `Inductive`.

```
CoInductive infList (A:Type) : Type :=
| nil : infList A
| cons : A → infList A → infList A.
```

Figure 5.19 – Definition of coinductive lists in Coq

```
Definition one_zero := cons 0 nil.

Fixpoint finite_zeros n :=
  match n with
  | 0 ⇒ nil
  | S n' ⇒ cons 0 (finite_zeros n')
end.

CoFixpoint infinite_zeros := cons 0 infinite_zeros.
```

Figure 5.20 – Examples of coinductive lists

An element of this type is either the empty list (`nil`) or the addition of an element to a possibly infinite list (`cons`).

Figure 5.20 shows three ways to define infinite lists. Finite lists can be defined using `Definition` or `Fixpoint`. Such definitions would also be valid definitions of standard inductive lists. The list `one_zero` is a list of one element, containing only 0. For any natural number `n`, `finite_zeros n` is the list of length `n` containing only 0. Infinite lists can be defined using `CoFixpoint`. `infinite_zeros` is an infinite list containing only 0.

The limitation that each finite prefix of a coinductive data has to be accessible in finite time imposes a syntactic restriction on the way `CoFixpoint` functions can be defined. In the definition of a `CoFixpoint` function, every corecursive call must be guarded by a constructor. This ensures that, for any natural number n, the n first elements of the object obtained by calling that `CoFixpoint` function are accessible in n recursive calls, and thus in finite time. The definition of `infinite_zeros` in Figure 5.20 comply with this rule. Indeed, the occurrence of `infinite_zeros` in the body is guarded by the constructor `cons`. Figure 5.21 shows two other tentative `CoFixpoint` definitions on infinite lists. The first one, `add_1` (Figure 5.21a), takes a possibly infinite list of natural numbers and increments each element by 1. The corecursive call `add_1 l'` is guarded by the constructor `cons`. This definition is thus a valid `CoFixpoint` definition. The second tentative definition, `filter` (Figure 5.21b), removes from the input list `l` the elements that do not satisfy the

```
CoFixpoint add_1 l :=
  match l with
  | nil ⇒ nil
  | cons n l' ⇒ cons (n+1) (add_1 l')
  end.
```

(a) Correct definition that increments each element of the input list `l` by 1

```
CoFixpoint filter (A : Type) (f : A → bool) (l : infList A) :=
  match l with
  | nil ⇒ nil
  | cons x l' ⇒ if f x then cons x (filter A f l') else filter A f l'
  end.
```

(b) Incorrect definition that removes from `l` the elements not satisfying `f`

Figure 5.21 – Tentative `CoFixpoint` definitions on infinite lists

predicate function `f`. The corecursive call in the then-branch is guarded by the constructor `cons`, but this is not the case of the corecursive call in the else-branch. Coq rejects this definition with the following message:

"Error: Recursive definition of `filter` is ill-formed. [..] Unguarded recursive call in `filter A f l'`."

The semantics of our programs is given as trajectories which are finite or (countably) infinite objects. Moreover, even when they are infinite, every finite prefix can be computed in finite time. Trajectories are thus good candidates for the use of coinductive types in Coq.

However, just like `filter` of Figure 5.21b, the projection operator Proj_L can remove an unpredictable number of consecutive elements in the trajectory on which it is applied. For example, when there is an infinite loop in the original program not preserved in the slice, the projection remove all the (label, state)-pairs produced by the loop, and there is an infinity of such pairs. This implies the impossibility to write the projection as a `CoFixpoint`, because such definition would not respect the guard condition of `CoFixpoint` functions.

To circumvent this problem, all the trajectories manipulated are finite. Propositions are then written using finite trajectories, but of unbounded sizes. For example, the function returning the trajectory of a program on a given input has type:

$$\texttt{traj_prog} : \texttt{nat} \to \texttt{state_eps} \to \texttt{prog} \to \texttt{traj}$$

where `prog` is the type of programs and `traj` := `list` (`label` ∗ `state_eps`) denotes lists of (label, state)-pairs. `traj_prog n s p` computes the `n` first elements of the trajectory of `p` on state `s`. If the trajectory of `p` on `s` is shorter than `n`, then it returns the whole trajectory.

5.5.5 Non-Uniqueness of Labels

On this thesis, we make the assumption that the labels in a program are all distinct. To avoid passing this assumption everywhere, it was not made in the Coq formalization. In practice, this means that statements with the same label are not distinguished by slicing. When one statement is preserved, then all the statements with the same label must be preserved, with their dependencies. As said above, this allows to have lighter lemmas and theorems. But it also has major drawbacks.

First, the non-uniqueness of labels implies that a subset of labels does not uniquely identify a quotient. This is thus not possible to define the slice as "the" quotient whose set of labels is the slice set. We conservatively define it as the largest quotient.

Second, it prevents from writing a function computing the projection. Indeed, the projection removes the elements whose labels are not in the set and projects the states of the remaining elements to the variables occurring in the statement that have generated them. Therefore, to compute the projection, it needs to retrieve a statement by its label. This is not possible *a posteriori* due to the non-uniqueness of labels, that is why we need to do it at the moment we generate the trajectory, and create a new function rather than reusing `traj_prog`. It is named `proj_traj_prog` and its type is the following:

$$\texttt{proj_traj_prog} : \texttt{nat} \rightarrow \texttt{state_eps} \rightarrow \texttt{prog} \rightarrow \texttt{partial_traj}$$

where

$$\texttt{partial_traj} = \texttt{list}\ (\texttt{label} * \texttt{partial_state_eps})$$

and `partial_state_eps` is either the error state or a partially defined state.

5.5.6 Formalization of Data Dependencies

Control and assertion dependence relations were relatively easy to formalize and manipulate, since they are rather simple and follow the structure of the program. They are thus well-suited for inductive reasoning. This is not the case of data dependence. Indeed, its formalization is the generic one, in terms of def-use paths (more precisely, data dependence is defined in the Coq formalization using a concept similar to finite syntactic paths (cf. Definition 5.3)). To handle

it, we wrote three auxiliary recursive functions (`mustdef_prog`, `maydef_prog` and `mayread_prog`) that compute parts of the data dependence information of a program. They allow to write an alternative definition of data dependence in the form of a recursive function, called `set_data_prog`, which computes data dependence in a structural way and is thus easier to manipulate in inductive proofs. This function is proved to correctly implement the inductive relation that defines data dependence in terms of def-use paths.

5.6 Related Work

This section presents works from the literature that are closely related to this chapter.

5.6.1 Debugging and Dynamic Slicing

The usage of slicing that is maybe the best established one is debugging. This application of slicing was even proposed by Mark Weiser [Wei82] soon after he proposed program slicing. Weiser suggests that programmers mentally use slice when debugging or understanding a program.

Korel et al. [KL88] and Agrawal et al. [ADS93] insist on the fact that static slicing is not adapted to debugging. Indeed, a bug is found on one or several specific inputs. One can use the information about these inputs to understand the bug, but static slicing does not take it into account. Dynamic slicing, instead, can take advantage of it and thus can be more precise than static slicing.

Hierons et al. [HHD99] propose to use static backward slicing in mutation testing. They claim that the time-consuming part of mutation testing is the classification of some mutants that are equivalent to the program or hard to kill, thus requiring human analysis. Static backward slicing can help classifying these mutants, either by automatically detecting that they are equivalent or by providing a simplified version to the user who can analyze it more easily.

While close to our topic, since it uses slicing in the context of a program with errors, these works focus on revealing an error, instead of proving their absence.

5.6.2 Slicing and Non-Termination

In Weiser's works [Wei81, Wei84], the link between the behaviors of the original program and of its slice is established only for inputs on which the execution of the original program terminates (cf. Theorem 4.1). In this restricted case, both behaviors are equivalent with respect to the slicing criterion.

Podgurski et al. [PC90] introduce the distinction between "strong control dependence", which is basically the standard control dependence also considered by Weiser, and "weak control dependence" which intuitively also takes into account the fact that loops that run indefinitely influences the statements following them, since they can prevent these statements from being executed (see Section 6.1 for related work about control dependence). They compare both notions with "semantic dependence", that models the intuitive notion of dependence between statements. In particular, they note that, since dependence based on strong control dependence fails to capture the semantic dependence induced by loops, it does not fully justify the usage of slicing proposed by Weiser for debugging. This work suggests that the classic soundness property of slicing does not hold with standard (strong) control dependence when non-termination is considered; and that, with more control dependencies (weak control dependence), the classic soundness property holds even in the presence of non-termination.

Three approaches have been explored in the literature to establish a soundness property for program slicing in the presence of non-termination:

- Changing the semantics of the language to capture exactly what is preserved by slicing
- Adding more dependencies, like the weak control dependence approach, so that the equivalence of behaviors still holds in the presence of non-termination
- Keeping the classic control dependencies, like the strong control dependence approach, and changing the soundness property

The third approach is the one adopted by relaxed slicing: keeping slices of reasonable sizes and weakening the soundness property. These three approaches are discussed below.

Changing the semantics. Some works are aware of the problems caused by infinite loops for the equivalence of behaviors of the program and its slices. Instead of modifying the slice or the semantic relation between the original program and its slice, they propose other semantics than the classic one that somehow continue execution even after an infinite loop.

Cartwright et al. [CF89] argue that the strict sequential semantics is too strict when considering optimizations. With the natural semantics, if the evaluation of an expressions diverges, or if a loop runs indefinitely, the whole resulting state becomes undefined. They propose two lazy semantics that uncouple the updates done on different variables. With these semantics, when the evaluation of a statement diverges, only part of the state becomes undefined, not the whole state. Especially, if a variable is not modified in a loop, its value after the loop is equal to its value before the loop, whether the loop terminates or not.

Giacobazzi et al. [GM03] and then Nestra [Nes09] note that slicing does not preserve non-termination (a slice may terminate when the initial program diverges), thus the standard denotational semantics is not adequate to model the link between a program and its slice. They propose transfinite generalizations of the semantics, that observe computations even after executing infinite loops.

Barraclough et al. [BBD+10] propose yet another semantics adapted to program slicing. Their semantics is strict. The program is given as semantics a family of traces indexed by a natural number. This natural number is the maximal number of iterations a loop can execute. When this bound is reached, the loop is left artificially, even if its condition is still evaluated to true. This is another way to observe the computations done by a program after an infinite loop. They prove that slicing preserves this semantics. They claim that it has several improvements compared to the previous ones described above. It is in particular intuitive and substitutive.

While these proposals are elegant, they are unsuitable for our purpose since they all consider somehow non-natural trajectories. It is unclear how these trajectories will combine with, for instance, path-oriented testing techniques like in [CKGJ12, GTXT11].

Adding more dependencies. The discrepancy between the non-termination behaviors of the program and its slice comes from some non-terminating statements that are sliced away. Adding more dependences to prevent these statements from being sliced away is a solution to establish the equivalence of behaviors in the presence of non-termination.

This is, for example, the approach of Ranganath et al. [RAB+07]. The start point of their reasoning is that control dependence definitions are not adapted to modern program structures manipulating exceptions or allowing non-termination on purpose, such as reactive systems. They propose two definitions of control dependence adapted to modern programs, a non-termination sensitive one (corresponding to the weak control dependence of [PC90]) and a non-termination insensitive one (corresponding to the strong control dependence of [PC90]). Moreover, they prove that slicing based on non-termination sensitive control dependence preserves the behavior of the original program by constructing a weak bisimulation between the executions of the initial program and its slice. This is discussed in more detail in Section 6.1.

Hatcliff et al. [HCD+99] use the same notion of weak bisimulation for slicing concurrent Java programs.

The limitations of these approaches is that slices are larger. Additional dependences mean additional statements in the slice. In the case of non-termination sensitive control dependence of [RAB+07], like for the case of weak control dependence of [PC90], this means preserving each loop that could be evaluated before

the slicing criterion. In our case with the addition of errors, this would mean preserving each loop or error-prone statement that could be evaluated before the slicing criterion. This does not seem realistic in the case of large programs.

Keeping the classic control dependencies. Instead of proposing new semantics or algorithms, some works have focused on describing the link between the program and its slices in terms of standard semantics.

Ball et al. [BH93] establish this link. In this work, they precisely describe a slicing algorithm in the presence of arbitrary control-flow for a structured language with breaks and gotos. They also prove its correctness, expressed as the replication by each statement in the slice of the behavior of the same statement in the original program. This replication is stated in two parts, in the same way as Theorem 5.1.

- For any state on which the initial program does not terminate normally, given a statement preserved in the slice, the sequence of values observed at this statement in the initial program is a prefix of that of the same statement in the slice.

- For any state on which the initial program terminates normally, given a statement preserved in the slice, the sequence of values observed at this statement in the initial program is identical to that of the same statement in the slice.

They make precise what they mean by "terminate normally": the execution terminates in a state called "EXIT", adding that the "execution can fail to terminate normally if the program includes an infinite loop or an exception such as division by zero". Apart from the difference of wording ("exception" instead of "error"), the difference with our work is:

- Theorem 5.1 manipulates the global trajectory of the program instead of the trajectory of values observed at each statement preserved in the slice. Actually, this preservation of the trajectory is contained in their intermediate results, but not in their final one.

- Contrary to their work, we apply this theorem in the context of verification and deduce from it an answer to the questions asked in Section 1.4.

Amtoft [Amt08] uses the framework of [RAB+07] and focuses on non-termination insensitive control dependence. It proves that the link between the behaviors of the initial program and its slice computed using this form of control dependence is a weak simulation, instead of a weak bisimulation in the case of non-termination sensitive control dependence [RAB+07]. This is equivalent to the formulation in terms of prefix used in this thesis (cf. Theorem 5.1).

We use the same approach as these works, but focus on the link between the program and its slice in terms of presence or absence of runtime errors. This justifies detecting runtime errors on the slices instead of the initial program.

5.6.3 Slicing in the Presence of Errors

Compared to the problem of non-termination, only a few works in the literature focus on slicing programs with possible errors.

Harman et al. [HSD96] focus on this topic. They note that classic algorithms only preserve a lazy semantics. To obtain correct slices with respect to a strict semantics, they propose to preserve all potentially error-prone statements by adding pseudo-variables in the $\mathsf{def}(l)$ and $\mathsf{ref}(l)$ sets of all potentially erroneous statements l. Our approach is more fine-grained in the sense that we can independently select assertions to be preserved in the slice. This benefit comes from our dedicated formalization of errors with assertions and a rigorous proof of soundness using a trajectory-based semantics. In addition, we make a formal link about the presence or the absence of errors in the program and its slices.

Harman et al. [HD95] use program slicing as well as meaning-preserving transformations to analyze a property of a program not captured by its own variables. They take the robustness of the accesses to an array as an example. They add variables and assignments in the same idea as our assertions. The slice is seen as an approximate answer to the question asked, which is often undecidable. If the slice is reduced to a single statement, this corresponds to the algorithm answering "yes" or "no". If the slice is larger, it is a more precise answer than "unknown". This is not clear, though, how they deal with multiple types of errors at the same time. Moreover, no formal justification is given.

Allen et al. [AH03] extend data and control dependences for Java program with exceptions. They are mostly focused on Java checked exceptions that are supposed to be caught by a calling function, and the impact it has on interprocedural control and data dependences. However, they also discuss unchecked exceptions, that are the equivalent to runtime errors in Java. They clearly make the connection with non-termination: "How unchecked exceptions should be treated in the context of slicing depends on the intended application. This question is related to the question of whether or not a statement S that follows a loop should be control dependent on the loop predicate". They also rule out the approach of making each statement following a statement that may raise an unchecked exception dependent on this statement, since this would lead to too large slices. Contrary to our work, no formal justification is given.

Rival [Riv05] designs static analyses to help the classification of alarms returned by an abstract interpreter called Astrée. He proposes to use static backward program slicing to apply these analyses on smaller programs. He uses a model that is

very similar to ours. His language is close to the WHILE language with assertions described in Section 5.1.1, but also includes some non-determinism in the form of an **input** statement. He also assumes that errors occur only after failed assertions. Moreover, he also proposes to use assertions as slicing criteria. However, he does not introduce new dependencies for assertions. Indeed, his expressions are supposed to be error-free, thus do not need to be protected like ours. A soundness property of slicing is given that compares sets of trajectories, not single trajectories, between the original program and its slice. This soundness property claims that each (label, state)-pair in the set of trajectories of the initial program can be associated to a (label, state)-pair in the set of trajectories of the slice. The order of execution is not considered in this soundness property, though, and thus the notion of prefix cannot be used. However, this result is strong enough to justify the application of the static analyses on the slice rather than on the initial program. In particular, Rival notes about this soundness property that: "The approximation [...] is strict in general, because slicing may remove causes for non-termination or errors, hence, introduce strictly more behaviors in the statements after **while** or **assert** statements." However, contrary to our work, no proof is given. Moreover, the analysis of the link between the presence or the absence of errors between a program and its slices is not precisely studied.

5.6.4 Certified Slicing

The ideas developed in [Amt08, RAB+07] were applied in [BMP15, Was11] using proof assistants.

Wasserrab et al. [WLS09, Was11] build a framework in Isabelle/HOL to formally prove a slicing defined in terms of graphs in the intraprocedural and interprocedural cases. Their formalization is independent both from the language and from the exact definition of control dependence in the intraprocedural case. They use the simulation-based proof of Amtoft [Amt08]. While far more advanced than our formalization, their development does not adopt the verification point of view under which we look at slicing. Moreover, it is really axiomatic and thus not so adequate to extract an executable slicer.

Blazy et al. [BMP15] propose an unproven but efficient slice calculator for an intermediate language of the CompCert C compiler [Ler09], as well as a certified slice validator and a slice builder written in Coq. The soundness property is also simulation-based, following Amtoft's [Amt08] and Wasserrab's [Was11] approach. The modeling of errors and the soundness of verification on slices is not specifically addressed in this work. Moreover, it is not clear whether the hypothesis that the program has a well-defined semantics, which is pervasive in CompCert, allows to use the soundness property established for verification, where input programs are supposed to be unsafe until analyzed.

This chapter presented relaxed slicing, i.e. a notion of slicing relying on standard control and data dependences even in the presence of errors (modeled as assertions) and non-termination, thus producing slices of reasonable sizes. An appropriate soundness property was established, that justifies the usage of slicing in verification chains, such as the two methods presented in Section 1.3: SANTE [CKGJ12] and the theory behind the SYMBIOTIC tool [SST12].

Relaxed slicing has been formalized in the Coq proof assistant on a representative structured language from which a certified program slicer can be extracted.

An obvious limit of this formalization of relaxed slicing is the choice of the language which is a WHILE language with errors. Abstracting from the language, like [Was11], seems a natural direction to extend the significance of our results. This is discussed in the next chapter.

Chapter 6

Formalization of Weak Control-Closure in Coq

The previous chapter proved that program slicing could be used for verification on a simple but representative language, namely a WHILE language with assertions which allows both infinite loops and errors modeled as failed assertions. One step further is to prove this in a richer language, e.g. one containing unstructured control flow such as **`goto`** statements. To progress into this direction, one would have to formalize dependence relations in a larger context. While in the previous chapters, the definition of data dependence was a generic one in terms of def-use paths (cf. Section 5.2.2), and thus could potentially be reused in a different context, control dependence was tightly coupled to the WHILE languages presented (cf. Sections 4.2.1 and 5.2.1). Indeed, the simple definition used completely relied on the fact that, in both languages, only **`if`** and **`while`** statements can introduce control dependence. To handle different and more complex languages, we can change the definition of control dependence, and adopt one of the language-independent definitions found in the literature.

We choose the definition given by Danicic et al. [DBH+11] that generalize control dependence on arbitrary finite directed graphs. We focus on the theory about non-termination insensitive control dependence (cf. Section 5.6) that we formalize and prove in the Coq proof assistant. This includes theoretical concepts, an algorithm computing from a given set the smallest superset closed under non-termination insensitive control dependence and a proof of the correctness of this algorithm.

This chapter is organized as follows. Section 6.1 presents the different forms of control dependence found in the literature, and in particular the definition given by Danicic et al. [DBH+11]. The next sections present in detail the theory of Danicic et al. about non-termination insensitive control dependence. They reflect the Coq formalization. Section 6.2 introduces weakly control-closed sets that

are sets closed under control dependence. Next, Section 6.3 proves the existence of weak control-closure, i.e. the smallest superset that is weakly control-closed. Section 6.4 illustrates on an example (the program of Figure 4.2) that Danicic et al.'s framework extends classic slicing by showing that the slice based on Danicic et al.'s weak control dependence is identical to the slice based on classic control dependence (cf. Figure 4.4). Then, Section 6.5 presents Danicic et al.'s algorithm computing weak control-closure along with its proof of correctness. Section 6.6 presents the Coq formalization and gives some observations about it.

Although all results presented in Sections 6.2, 6.3 and 6.5 of this chapter are mechanically proved in Coq (see Section 6.6), we present the paper-and-pencil version of the proofs in order to make this chapter complete. The proofs that have a mechanized counterpart in the Coq formalization [Léc18] are marked as such. Like the proofs of Chapter 5, the paper-and-pencil proofs of this chapter are not necessarily structured in the same way as their Coq counterparts, even though they prove the same statements.

6.1 State of the Art about Control Dependence

Control dependence informally characterizes the influence of certain statements on the execution of other statements in a program. If the evaluation of s_1 decides whether s_2 is executed afterwards or not, then s_2 is said to be control dependent on s_1.

6.1.1 Structured Control Flow

In structured programs, like those manipulated in Chapters 4 and 5, control dependence is defined structurally (cf. Definition 5.2) and algorithms computing control dependence can be just syntax analyzers [KL88].

6.1.2 Unstructured Control Flow Using Control Flow Graphs

When programs with unstructured control flow are considered, this is more complicated. The literature came up with definitions of control dependence that are language-independent and applicable on control flow graphs [ASU86, All70] (CFGs) representing programs. An hypothesis is made on these CFGs: there exists a unique exit node[1] reachable by all the other nodes. In this case, control dependence can be defined using post-domination, as explained hereafter.

[1]We will use node or vertex indifferently in this thesis to denote points in a graph.

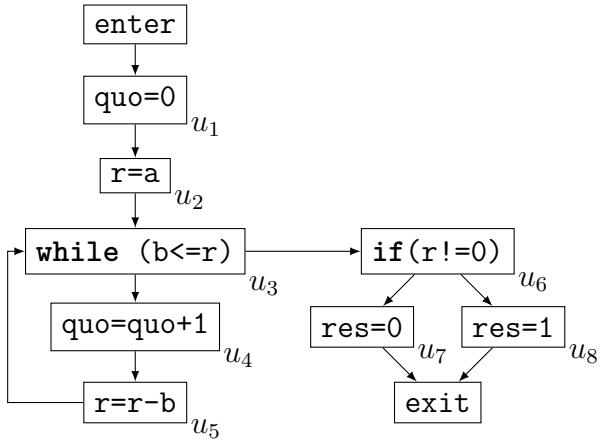


Figure 6.1 – Control flow graph G of the program of Figure 4.2

To illustrate the subsequent definitions, we introduce the CFG G of program p testing the divisibility using euclidean division introduced in Figure 4.2. It is given in Figure 6.1. Each statement of p is associated to a node in the CFG and is given a label. Two nodes are added: a node `enter` representing the start of an execution that is connected to the first statement (u_1) and a node `exit` representing the end of an execution that is connected to the possible last statements executed (u_7 and u_8). Edges in the CFG indicate the possible control flow transfers:

- Assignments (u_1, u_2, u_4, u_5, u_7 and u_8) can transfer the control flow only to the next statement.
- Conditionals (u_6) and loops (u_3) introduce branching. The conditional in node u_6 can transfer the flow either to the then-branch (u_7) or to the else-branch (u_8). The loop in node u_3 can lead to its body (u_4) or to the next statement (u_6).

Note that `exit` is a unique exit vertex reachable by every other node, this CFG is thus appropriate to illustrate control dependence based on post-domination.

Denning et al. [DD77] were the first to introduce control dependence (that they call "implicit flow") in the presence of unstructured control flow using the notion of post-dominator (that they call "forward dominator").

To clearly describe Denning et al.'s work, we first introduce post-domination and one of its well-known properties.

Definition 6.1: Post-dominator

A node n_2 post-dominates a node n_1 if it lies on every path from n_1 to the exit vertex.

We choose to allow that a node post-dominates itself, like in e.g. [PC90]. There also exist definitions where this is not allowed (e.g. [DD77]).

Considering the CFG of Figure 6.1, u_2 post-dominates u_1, u_5 post-dominates u_4, u_6 post-dominates u_4 and `exit` post-dominates u_7. u_4 does not post-dominate u_2, since there exists a path $(u_2, u_3, u_6, u_7, \texttt{exit})$ from u_2 to `exit` that does not traverse u_4. Likewise, u_7 does not post-dominate u_2 nor u_6.

For each node different from the exit vertex, there exists a post-dominator that is closer to this node than any other post-dominator [ASU86, PC90].

Property 6.1

On the paths from a given node u, distinct from the exit vertex, to the exit vertex, the first post-dominator encountered that is not u itself is independent of the path considered. It is called the immediate post-dominator of u.

Proof. We give a sketch of the proof [ASU86]. Actually, we first prove a stronger result about acyclic paths. The post-dominators of u always occur in the same order on all the acyclic paths from u to the exit vertex. Let v_1 and v_2 be two distinct post-dominators of u and assume that there exist two acyclic paths π_1 and π_2 from u to the exit vertex such that v_1 occurs before v_2 on π_1 and v_2 occurs before v_1 on π_2. If we concatenate the prefix of π_1 from u to v_1 and the suffix of π_2 from v_1 to the exit vertex, we obtain a path from u to the exit vertex that does not contain v_2, which contradicts that v_2 is a post-dominator of u.

Let π_1 and π_2 be two paths from u to the exit vertex and let v_1 and v_2 be the first post-dominators occurring on π_1 and π_2 respectively. Assume that v_1 and v_2 are distinct. Since v_2 post-dominates u, it also occurs on π_1, but after the first occurrence of v_1. From π_1, it is possible to construct an acyclic path π_1' from u to the exit vertex such that v_1 occurs before v_2 on it. Likewise, from π_2, it is possible to construct an acyclic path π_2' from u to the exit vertex such that v_2 occurs before v_1 on it. But, according to what we have just proved about acyclic paths, this is contradictory. □

Denning et al. define, given a conditional node u, the set of nodes affected by u as the set of nodes occurring on a path from u to its immediate post-dominator, where the immediate post-dominator occurs only at the end, and that are not u or its immediate post-dominator.

Definition 6.2

A node n_2 is control dependent on a node n_1 if n_2 is distinct from n_1 and from the immediate post-dominator of n_1, and there exists a path from n_1 to its immediate post-dominator, containing the immediate post-dominator only at its end, that contains n_2.

Applying this definition on the CFG of Figure 6.1, we can deduce from the path u_3, u_4, u_5, u_3, u_6 connecting u_3 to its immediate post-dominator u_6 that u_4 and u_5 are control dependent on u_3. Likewise, from the paths $u_6, u_7, \texttt{exit}$ and $u_6, u_8, \texttt{exit}$ that connect u_6 to its immediate post-dominator `exit`, we can deduce that u_7 and u_8 are control dependent on u_6. This definition gives therefore the same control dependencies than the structural one (cf. Section 5.2.1).

Weiser [Wei84] defines slicing based on the same characterization of control dependence as Denning et al. [DD77], but calls post-dominators "inverse dominators".

Ferrante et al. [FOW87] use another definition of control dependence when defining the program dependence graph. It is also based on post-domination but not equivalent to Denning et al.'s definition [DD77].[2]

Definition 6.3

A node n_2 is said to be control dependent on a node n_1 if n_2 does not post-dominate n_1 and there exists a path connecting n_1 and n_2 such that each node on this path except n_1 and n_2 is post-dominated by n_2.

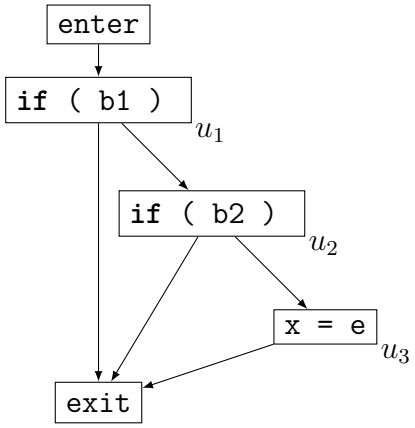
Let us illustrate this second definition using the CFG of Figure 6.1. u_4 does not post-dominate u_3 and it is vacuously true that on the path u_3, u_4 all the nodes except u_3 and u_4 are post-dominated by u_4, thus u_4 is control dependent on u_3. We can show that u_5 is also control dependent on u_3, and that u_7 and u_8 are control dependent on u_6. This new definition of control dependence gives therefore the same dependencies as Denning et al.'s and Weiser's definition [DD77, Wei84] in this example.

This is not true in the general case though. Consider the program represented in Figure 6.2. According to Denning et al.'s definition [DD77], the statement u_3 is control dependent on the conditional u_1. Indeed, u_3 occurs on the path $u_1, u_2, u_3, \texttt{exit}$ that connects u_1 to its immediate post-dominator `exit`. On the contrary, u_3 is

[2] Definition 6.3 makes loop conditions not dependent on themselves, while they are in [FOW87]. The difference comes from the fact that in [FOW87], a node cannot post-dominate itself.

```
1 if ( b1 ) {
2     if ( b2 ) {
3         x = e;
4     }
5 }
```

(a) Source code



(b) CFG

Figure 6.2 – An example program with nested conditionals and its CFG

not control dependent on u_1 using Ferrante et al.'s definition [FOW87]. Indeed, the only path from u_1 to u_3 is u_1, u_2, u_3 and u_2 does not post-dominate u_1. Both definitions are thus not strictly equivalent.

However, one can show that, in Ferrante et al.'s definition [FOW87], u_3 is control dependent on u_2 which is itself control dependent on u_1. This means that u_3 is not directly control dependent on u_1, but it is transitively. Ferrante et al. [FOW87] observe that Denning et al.'s and Weiser's definition [DD77, Wei84] is equivalent to the transitive closure of their definition.

Ferrante et al.'s definition [FOW87] can be stated in an equivalent way (e.g. [PC90, AH03, Was11, DBH+11]). A node n_2 is said to be control dependent on a node n_1 if $n_1 \neq n_2$ and n_1 has two children n_1' and n_1'' such that n_2 post-dominates n_1' but not n_1''.

In the example CFG of Figure 6.1, u_3 has two children u_6 and u_4. u_4 post-dominates itself but does not post-dominate u_6, it is therefore control dependent on u_3.

Podgurski et al. [PC90] introduce two kinds of control dependence (already discussed in Section 5.6): strong control dependence, equivalent to Denning et al.'s and Weiser's definition, and weak control dependence that also takes into account control dependence due to possibly non-terminating loops. Weak control dependence is defined using "strong forward domination" (Podgurski et al. use the term "forward" like Denning et al. [DD77] to denote post-domination).

Definition 6.4

A vertex n_2 strongly forward dominates a vertex n_1 if n_2 forward dominates (i.e. post-dominates) n_1 and every path from n_1 that is longer than a given size contains n_2.

The second part of the definition requires that n_1 will reach n_2 in finite time. In particular, if n_2 strongly forward dominates n_1, there is no infinite loop that could prevent the execution from reaching n_2 from n_1.

In the CFG shown in Figure 6.1, u_6 does not strongly forward dominate u_3 since there exists a path that does not leave the loop and thus never reaches u_6.

Podgurski et al. define weak control dependence as follows, in a similar way to the variant of Ferrante et al's definition [FOW87] presented above.

Definition 6.5

A node n_2 is directly weakly control dependent on a node n_1 if n_1 has two children n_1' and n_1'' such that n_2 strongly forward dominates n_1' but not n_1''. Weak control dependence is defined as the transitive closure of direct weak control dependence.

In the CFG of Figure 6.1, u_6 is directly weakly control dependent on u_3. Indeed, u_3 has two children u_4 and u_6, u_6 does not strongly forward dominate u_4 due to the loop in u_3, but strongly forward dominates itself. u_3, u_4 and u_5 are also directly weakly control dependent on u_3. Vertices u_7 and u_8 are directly weakly control dependent on u_6 which is directly weakly control dependent on u_3, they are thus (indirectly) weakly control dependent on u_3.

Bilardi et al. [BP96] propose to parametrize the definition of domination, and thus the definition of control dependence, with a set of paths, giving rise to an infinite number of variants of control dependence. They prove that Podgurski et al.'s strong and weak control dependence relations [PC90] can be encoded in their framework.

Algorithms computing control dependence based on post-domination compute the "post-dominator tree" that stores efficiently post-domination information (e.g. [FOW87]). The best-known algorithm that computes domination information is probably the almost linear algorithm of Lengauer et al. [LT79]. Further ameliorations of this algorithm were proposed (e.g. [CHK01, GTW06, FGMT13]), and even certified ones (e.g. [BDP15]). Since these algorithms computing domination information are very efficient, algorithms computing control dependence that are based on them are efficient too.

6.1.3 Unstructured Control Flow on Arbitrary Graphs

All the definitions of control dependence that have been just given rely on the hypothesis that the CFG has a unique exit node.

Ranganath et al. [RAB+07] observe that this hypothesis is not realistic any more when recent programs, that heavily use exception-processing and that may purposely non-terminate, are considered. This is especially the case of reactive programs whose normal execution is infinite. They introduce non-termination sensitive control dependence that is proved to extend Podgurski et al.'s weak control dependence [PC90] on CFGs without a unique end node. This form of control dependence is called "non-termination sensitive" since it takes into account control dependence due to potentially non-terminating statements. They prove that slicing based on this form of control dependence preserves the behavior of the initial program using a weak bisimulation between the executions of the original program and its slice. They also introduce non-termination insensitive control dependence that ignores control dependence due to potentially non-terminating statements. It is proved to extend classic forms of control dependence (in particular Podgurski et al.'s strong control dependence [PC90]) on CFGs without a unique end node. They did not prove the correctness of this form of slicing.

Amtoft [Amt08] completes Ranganath et al.'s formalization by proving the correctness of slicing based on non-termination insensitive control dependence: the link between the program and its slice is a weak simulation, not a weak bisimulation. This formalizes the idea that the behavior of the original program is a prefix of the behavior of the slice.

Danicic et al. [DBH+11] design a framework for control dependence that removes all the constraints on CFGs and work with arbitrary directed graphs. They prove that it subsumes all previous formalizations of control dependence in that it coincides with former definitions of control dependence on the kinds of CFGs on which these former definitions are defined.

More precisely, it introduces two kinds of control dependence:

- weak control dependence that is non-termination insensitive[3] and subsumes former non-termination insensitive control dependence definitions (Denning et al.'s [DD77], Ferrante et al.'s [FOW87] and Amtoft's [Amt08] presented above),

- strong control dependence that is non-termination sensitive and subsumes former non-termination sensitive control dependence definitions (Podgurski et al.'s weak control dependence [PC90] and Ranganath et al.'s [RAB+07] presented above).

[3]Note that the terminology in terms of "weak" and "strong" is the opposite of Podgurski et al.'s [PC90]

Their approach is quite different from previous works. Instead of characterizing the relations between nodes, they rather directly characterize the fact of being closed under control dependence. They define weakly (resp. strongly) control-closed sets that formalize sets closed under weak (resp. strong) control dependence. They prove the existence of weak (resp. strong) control-closure that is the smallest superset weakly (resp. strongly) control-closed. They provide two algorithms computing weak and strong control-closures and prove them correct.

Besides, they also give a semantics for each form of control dependence in terms of projections of paths in the graph that resembles the soundness properties established for slicing.

Two recent works of Amtoft reuse Danicic et al.'s framework for slicing. In [AAC13], Amtoft et al. study the slicing of extended finite state machines. In [AB16], Amtoft et al. focus on the slicing of probabilistic programs. In both works, an algorithm computing weak control-closure is designed and integrated in a rather efficient slicing algorithm.

Danicic et al.'s work is particularly interesting in the perspective of a generic slicer, since it unifies all previous control dependence definitions and provides algorithms.

As explained in Section 5.6, in this thesis we opt for non-termination insensitive control dependence that allows to create slices of reasonable sizes and corresponds to classic algorithms. Thus, from now on, we ignore strong control dependence and focus only on weak control dependence.

In Danicic et al.'s work about weak control dependence, three results are of particular interest for us:

- the definition of weak control-closed sets of vertices;
- the proof of existence of weak control-closure;
- the algorithm computing weak control-closure and its proof of correctness.

To ensure the same level of confidence in Danicic et al.'s result as in the results of Chapter 5, we formalize the three points above in the Coq proof assistant, slightly adapting some of Danicic et al.'s definitions of certain basic objects in order to be closer to the definitions used in other works of the literature. These small changes do not impact the meanings of the elaborated definitions such as the one of weak control-closed set or weak control-closure. They are unchanged.

Next sections present in detail some elements of Danicic et al.'s formalization of weak control dependence.

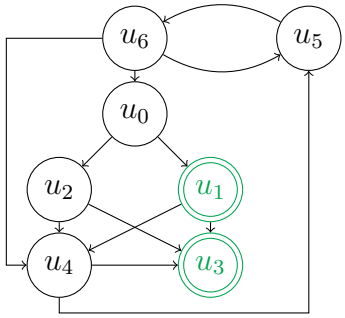


Figure 6.3 – Example graph G_0, with $V_0' = \{u_1, u_3\}$

6.2 Weak Control-Closed Sets

In this section and all the subsequent ones, $G = (V, E)$ denotes a finite directed graph with finite set of vertices V and set of edges $E \subseteq V \times V$, and V' denotes a subset of V. G must be thought of as the program control flow graph and V' as the slicing criterion, in a form of program slicing where only control dependence would be used as dependence relation.

We will use as a running example, in this chapter and the following ones, the finite graph $G_0 = (V_0, E_0)$ shown in Figure 6.3, with considered subset of vertices $V_0' = \{u_1, u_3\}$. G_0 is deliberately contrived to illustrate on a simple example the multiple concepts defined hereafter. Note in particular that it does not have any entry nodes or end nodes to highlight the fact that the upcoming definitions can be applied on any finite directed graph.

Here are two basic preliminary definitions before really entering the world of weak control dependence.

Definition 6.6: Path

We say that there is a *path* in G between two vertices u and v, denoted $u \xrightarrow{path} v$, if there exist $n \geq 0$ and vertices (in V) $u_0 = u, \ldots, u_n = v$ such that

$$\forall i, 0 \leq i < n \implies (u_i, u_{i+1}) \in E$$

In this case, we say that the path is of length n.

In G_0 shown in Figure 6.3, u_5, u_6, u_5, u_6, u_0 is a path of length 4. u_0 is a path of length 0, also referred as trivial.

Definition 6.7: Reachable nodes

The set of vertices in G reachable from a subset V' of vertices is denoted $\mathsf{R}_G(V') = \{v \in V \mid \exists u \in V',\, u \xrightarrow{path} v\}$.

In G_0 presented in Figure 6.3, $\mathsf{R}_{G_0}(\{u_3\}) = \{u_3\}$, since u_3 is not the source of any edges. $\mathsf{R}_{G_0}(V_0') = V_0$, since except u_3 every node can reach every other node. Since every vertex in G_0 is reachable from V_0', we will omit in the following examples the parts of the definitions requiring reachability, since they are always verified.

The first definition that we introduce is the one of V'*-path*. It designates paths that reach V' and ends as soon as they enter V'.

Definition 6.8: V'-path

A path π in G is said to be a V'*-path* in G if the last vertex of the path is in V' and all the other vertices in π are not in V'. In particular, if $u \in V'$, the only V'-path starting from u is the trivial path u. The existence of a V'-path between two nodes u and v is denoted $u \xrightarrow{V'-path} v$.

For example, in our example graph G_0 of Figure 6.3, u_1 and u_3 are trivial V_0'-paths. u_5, u_6, u_0, u_1 is a V_0'-path. u_0, u_1, u_4 and u_0, u_1, u_3 are not V_0'-paths; indeed, since u_1 is in V_0', it should have been the last vertex of the paths.

Note that, if we consider another subset V_1' verifying $V_1' \subseteq V'$, then every V'-path ending in V_1' is a V_1'-path. This property will be used multiple times in proofs in the rest of this document.

Based on the definition of V'-path, we can build the concept of *observable vertices*, following the naming of other works in the literature. Ranganath et al. [RAB+07] uses indeed the term "first observable elements", Wasserrab [Was11] defines "observable sets", while Blazy et al. [BMP15] designates such vertices as "next observable vertices". On the contrary, Danicic et al. only designate them using a function Θ. Observable vertices from a node u in V' are the nodes in V' first-reachable from u. We can give a simple definitions using V'-paths.

Definition 6.9: Observable vertex

Let $u \in V$. A vertex is an *observable vertex* from u in V' if it is the end of a V'-path from u. The set of observable vertices from u in V' is denoted $\mathsf{obs}_G(u, V')$, i.e. $\mathsf{obs}_G(u, V') = \{u' \in V \mid u \xrightarrow{V'-path} u'\}$.

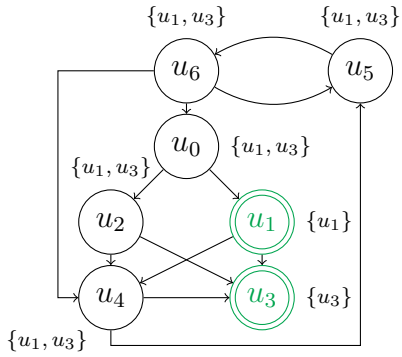


Figure 6.4 – Example graph G_0 annotated with the set of observable vertices with respect to $V_0' = \{u_1, u_3\}$

Note that, given a vertex u in V', since the only V'-path from it is the trivial path u, the set of observable vertices from it in V' is the singleton containing only itself, i.e. $\mathsf{obs}_G(u, V') = \{u\}$.

Using our running example G_0 of Figure 6.3,

$$\mathsf{obs}_{G_0}(u_1, V_0') = \{u_1\} \text{ and } \mathsf{obs}_{G_0}(u_3, V_0') = \{u_3\}$$

The set of observable vertices in V_0' from every other node is V_0'. For instance, u_0, u_1 and u_0, u_2, u_3 are two V_0'-paths, thus $V_0' \subseteq \mathsf{obs}_{G_0}(u_0, V_0')$. Since, $\mathsf{obs}_{G_0}(u_0, V_0') \subseteq V_0'$, $\mathsf{obs}_{G_0}(u_0, V_0') = V_0'$. Figure 6.4 shows G_0 where each vertex is annotated with the set of their observable vertices in V_0'.

The definition of V'-path is not the same as the one used in Danicic et al.'s work. Danicic et al.'s definition requires the path to be at least of length 1, thus rejecting trivial paths, and does not constrain the first vertex that may or may not be in V'. With this definition, $u_1, u_4, u_5, u_6, u_0, u_1$ would be a V_0'-path in G_0 shown in Figure 6.3. As stated above, the definition was changed so that observable vertices could be defined in a simple way from V'-paths and in a similar way to other works using them in the literature.

We now introduce the concept of *V'-weakly committing vertices*. These are vertices that, if they can lead to V', always lead to the same node in V'. Vertices that cannot lead to V' are automatically V'-weakly committing.

Definition 6.10: V'-weakly committing vertex

A vertex u in G is *V'-weakly committing* if all the V'-paths from u have the same end point (in V'). In particular, any vertex $u \in V'$ is V'-weakly committing. The set of V'-weakly committed nodes in G is denoted $\mathsf{WC}_G(V')$.

This definition can be rephrased using the concept of observable vertex. A vertex u in G is V'-weakly committing if and only if $|obs_G(u, V')| \leq 1$, where $|.|$ is the classic notation for the cardinal of sets.

In G_0 depicted in Figure 6.3, u_1 and u_3 belong to V_0' and thus are V_0'-weakly committing. As stated above, all vertices except u_1 and u_3 have a set of observable vertices in V_0' equal to V_0'. Since $|V_0'| = 2$, all theses vertices are not V'-weakly committing.

A node u which is not V'-weakly committing is the source of two V'-paths leading to two different vertices in V'. Intuitively, this is a decision node outside the slice that can lead to two distinct points in the slice. It has an impact on the slice but is itself not in the slice. This is the kind of nodes that prevent the slice from being closed under control dependence.

We can finally introduce the notion of weakly control-closed set.

Definition 6.11: Weakly control-closed set

A subset V' of V is *weakly control-closed* in G if every vertex reachable from V' is V'-weakly committing.

In the rest of the document, we will not always specify in which graph the subset is weakly control-closed when this is obvious.

Note that this definition is slightly different from Danicic et al's. This difference is inherited directly from the different definitions of V'-paths. Indeed, in Danicic et al.'s work, V' is V'-weakly committing if and only if every vertex *not in* V' and reachable from V' is V'-weakly committing. With our different definition, nodes in V' are trivially V'-weakly committing, thus we can simplify a bit Danicic et al.'s definition.

In our example graph G_0, all nodes are reachable from u_1 thus from V_0'. Thus every vertex must be V_0'-weakly committing for V_0' to be weakly control-closed in G_0. But all the nodes not in V_0' (u_0, u_2, u_4, u_5, u_6) are not V_0'-weakly committing. V_0' is therefore clearly not weakly control-closed in G_0. But other subsets of V_0 are weakly control-closed in G_0. The empty set $\emptyset$, the whole set of vertices V_0 and every singleton are weakly control-closed in G_0 by definition. More interesting weakly control-closed sets in G_0 include $\{u_0, u_1\}$, $\{u_0, u_5, u_6\}$, $\{u_0, u_1, u_4, u_6\}$ and $\{u_0, u_1, u_2, u_3, u_4, u_6\}$.

This definition of weakly control-closed set confirms the intuition given about V'-weakly committing vertices. These vertices are an issue when they are also reachable from V'. But it does not give a way, from an arbitrary V', to construct the smaller weakly control-closed superset. Intuitively, it specifies what is a correct slice, but does not give a method to construct it from the slicing criterion. Since reachable non-V'-weakly committing vertices are the problem, the naive solution

would be to add all these vertices to V'. While correct, this does not produce the smaller slice. In Section 6.3, we will show that we can produce smaller slices using another characterization of weakly control-closed sets.

6.3 Weak Control-Closure

In this section, the problem we consider is the following. Given an arbitrary subset V' of V, can we and how can we construct the smaller superset of V' that is weakly control-closed in G, i.e. what nodes should we add to V' to turn it into a weakly control-closed subset? We suggested to add the seemingly problematic nodes, i.e. the vertices both reachable from V' (i.e. in $\mathsf{R}_G(V')$) and not V'-weakly committing (i.e. in $V \setminus \mathsf{WC}_G(V')$). Lemma 6.1 states that this intuition is correct.

Lemma 6.1

$V' \cup \Big(\mathsf{R}_G(V') \cap (V \setminus \mathsf{WC}_G(V'))\Big)$ is weakly control-closed in G.

Proof.[4] Let $W = V' \cup \Big(\mathsf{R}_G(V') \cap (V \setminus \mathsf{WC}_G(V'))\Big)$. Let us prove that W is weakly control-closed using Definition 6.11 and Definition 6.10. Let $u \in V$ reachable from W. Let us show that u is W-weakly committing. For that, assume that there are two W-paths π_1 and π_2 from u reaching vertices v_1 and v_2 respectively in W and show that $v_1 = v_2$.

First, note that u is reachable from V'. Indeed, by definition of W, every vertex in W is reachable from V'. Since u is reachable from W, by transitivity it is reachable from V'.

If one of the W-paths were trivial, we would have $u = v_1 = v_2$. Assume that both W-paths are not trivial. In this case, $u \notin W$, i.e. $u \notin V'$ and either $u \notin \mathsf{R}_G(V')$ or $u \in \mathsf{WC}_G(V')$. Since $u \in \mathsf{R}_G(V')$, we have $u \notin V'$ and $u \in \mathsf{WC}_G(V')$.

Both v_1 and v_2 belong to a union. Consider the case where one of them is in the set $\mathsf{R}_G(V') \cap (V \setminus \mathsf{WC}_G(V'))$. Assume without loss of generality that $v_1 \in \mathsf{R}_G(V') \cap (V \setminus \mathsf{WC}_G(V'))$. This means that there are two V'-paths π_{11} and π_{12} from v_1 ending in v_{11} and v_{12} respectively such that $v_{11} \neq v_{12}$. By prepending π_1 to π_{11} and π_{12} respectively, we can construct two V'-paths from u ending in v_{11} and v_{12}. Again, since u is V'-weakly committing, this means that $v_{11} = v_{12}$, contradicting the hypothesis that they are distinct vertices.

Thus neither v_1 nor v_2 is in $\mathsf{R}_G(V') \cap (V \setminus \mathsf{WC}_G(V'))$. This means that both v_1 and v_2 are in V'. π_1 and π_2 are thus not only W-paths, they are also V'-paths. Since u is V'-weakly committing, we can deduce $v_1 = v_2$.

[4] The mechanized version of this proof is available in [Léc18].

This proves that u is W-weakly committing, which means that W is weakly control-closed. □

Lemma 6.1 gives a first method to build a non-trivial superset of V' that is weakly control-closed (V was indeed already known as a weakly control-closed superset of V', but it is not a very interesting one). Is it an optimal construction? Let us consider our example depicted in Figure 6.3. Applying Lemma 6.1 to V'_0 gives us V_0 as weakly control-closed superset of V'_0. But another weakly-control superset of V'_0 was presented earlier: $\{u_0, u_1, u_2, u_3, u_4, u_6\}$. This superset is strictly smaller than V_0, which proves that Lemma 6.1 does not provide an optimal construction.

Vertex u_5 is wrongly added by Lemma 6.1, in the sense that, without it, the set would already have been weakly control-closed. Vertex u_5 is reachable from V' and not V'-weakly committing, like u_0, u_2 and u_6, but what differentiates u_5 from the other vertices is that every V'-path from u_5 goes through u_6. Therefore, if u_6 is selected, u_5 automatically has only one observable vertex in the resulting set W and becomes W-weakly committing. We need a different notion that filters out u_5 but keeps the other vertices. We introduce the notion of V'-*weakly deciding vertex*.

Definition 6.12: V'-weakly deciding vertex

A vertex u in G is V'-*weakly deciding* if there exist two non-trivial V'-paths from u that share no vertex except u. Let $\mathsf{WD}_G(V')$ denote the set of V'-weakly deciding vertices in G.

Definition 6.12 is again different than the equivalent one in [DBH+11], due to the different notion of V'-path. The most visible change is that in our formalization, V'-weakly deciding cannot be in V', while this is possible in [DBH+11].

Property 6.2

$\mathsf{WD}_G(V') \cap V' = \emptyset$.

Proof.[5] This is immediate from Definitions 6.8 and 6.12. Since the V'-paths are non-trivial, this means that their start vertex, i.e. u, is not in V'. □

In our example graph G_0 of Figure 6.3, u_1 and u_3 are not V'_0-weakly deciding, due to Property 6.2. u_6 is V'_0-weakly deciding, with associated V'-paths u_6, u_0, u_1 and u_6, u_4, u_3. These V'-paths are highlighted in Figure 6.5. Likewise, u_0, u_2 and u_4 are V'_0-weakly deciding. On the contrary, u_5 is not V'_0-weakly deciding, since every V'_0-path from u_5 traverses u_6. Thus, $\mathsf{WD}_{G_0}(V'_0) = \{u_0, u_2, u_4, u_6\}$.

[5] The mechanized version of this proof is available in [Léc18].

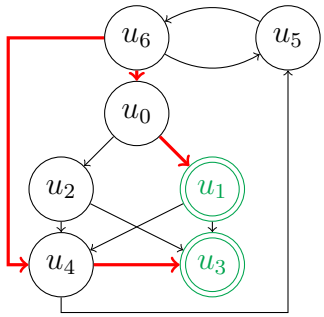


Figure 6.5 – Example graph G_0, with $V_0' = \{u_1, u_3\}$, and u_6 highlighted as a V_0'-weakly deciding vertex

This notion of V'-weakly deciding vertex can be used to give an alternative definition to weakly control-closed subsets.

Lemma 6.2

V' is weakly control-closed in G if and only if there is no V'-weakly deciding vertex in G reachable from V'.

Proof.[6] If V' is weakly control-closed in G, then by Definition 6.11, every vertex reachable from V' is weakly-committing, thus not weakly-deciding.

Reciprocally, assume that there is no V'-weakly deciding vertex reachable from V'. Let us also assume that there is a vertex u both reachable from V' and not V'-weakly committing, i.e. with at least two distinct observable vertices v_1 and v_2 in V', and deduce a contradiction. There exist two V'-paths π_1 and π_2 from u ending in v_1 and v_2 respectively. Let us define v as the last intersection on π_1 of π_1 and π_2. v is the source of two non-trivial V'-paths, the suffixes of π_1 and π_2, that share no vertex except v, thus v is V'-weakly deciding. Moreover, v is reachable from u which is itself reachable from V'. v is therefore reachable from V'. Thus, by Definition 6.12, v is V'-weakly deciding, which gives us the desired contradiction. This gives us that every vertex reachable from V' is V'-weakly committing, which means by Definition 6.11 that V' is weakly control-closed. □

In Figure 6.5, vertex u_6 is highlighted as a V_0'-weakly deciding vertex in G_0. By Lemma 6.2, this gives another proof that V_0' is not weakly control-closed in G_0.

Intuitively, Lemma 6.2 shows that V'-weakly deciding vertices reachable from V' are good candidates to be added to V' to construct the weak control-closure of V'. And since there are fewer V'-weakly deciding vertices reachable from V' than non-V'-weakly committing vertices reachable from V', we can hope to get this time

[6] The mechanized version of this proof is available in [Léc18].

the minimality that we did not have in Lemma 6.1. Theorem 6.1 shows indeed that this is the case, but we first prove several auxiliary results about V'-weakly deciding vertices.

Property 6.3

$$\forall\ V_1', V_2' \subseteq V,\ V_1' \subseteq V_2' \implies \mathsf{WD}_G(V_1') \subseteq V_2' \cup \mathsf{WD}_G(V_2')$$

Proof.[7] Let V_1' and V_2' two subsets of V such that $V_1' \subseteq V_2'$, and let $u \in \mathsf{WD}_G(V_1')$. If $u \in V_2'$, then trivially $u \in V_2' \cup \mathsf{WD}_G(V_2')$. Assume that $u \notin V_2'$ and prove that $u \in \mathsf{WD}_G(V_2')$. There exist two V_1'-paths π_1 and π_2 from u. Since $V_1' \subseteq V_2'$, π_1 and π_2 both have a vertex in V_2'. Let v_1 be the first vertex on π_1 in V_2' and v_2 the first vertex on π_2 in V_2', and let ρ_1 and ρ_2 the prefix of π_1 ending in v_1 and the prefix of π_2 ending in v_2 respectively. π_1 and π_2 are two V_2'-paths from u. Moreover, their intersection is equal to $\{u\}$, since they are the prefixes of paths whose intersection is equal to $\{u\}$. By Definition 6.12, u is V_2'-weakly deciding, which ends the proof. □

Property 6.4

$$\mathsf{WD}_G(V' \cup \mathsf{WD}_G(V')) = \emptyset.$$

The proof of Property 6.4 turned out to be, if not difficult, at least very long. The proof of the corresponding result [DBH+11, Lemma 53] in Danicic et al.'s work is quite compact, reasoning by contradiction, but it appeared to be inaccurate. We followed a different path. We state an auxiliary result and prove it by exploring all the possible sub-cases and showing that they can be transformed into the standard case.

We first introduce a new definition to manipulate weak variants of V'-path. These are V'-paths but with the constraint on the last vertex removed. This definition will reappear in Chapter 8.

Definition 6.13: V'-disjoint path

A path π in G is said to be a *V'-disjoint path* in G if all the vertices in π except the last one are not in V'. The existence of a V'-disjoint path between two vertices u and v is denoted $u \xrightarrow{V'-disjoint} v$.

In G_0 shown in Figure 6.3, u_6, u_0 is a V'-disjoint path.

A V'-path is a particular case of a V'-disjoint path. Given $V_1' \subseteq V'$, every V'-disjoint path is also a V_1'-disjoint path.

[7] The mechanized version of this proof is available in [Léc18].

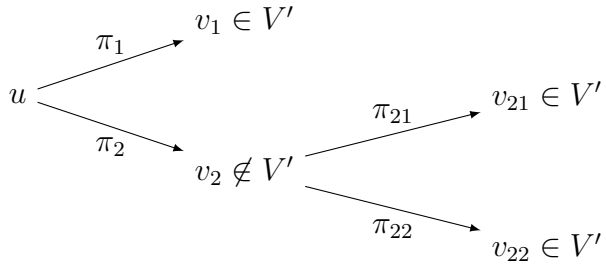


Figure 6.6 – Schematic representation of the configuration of Lemma 6.3

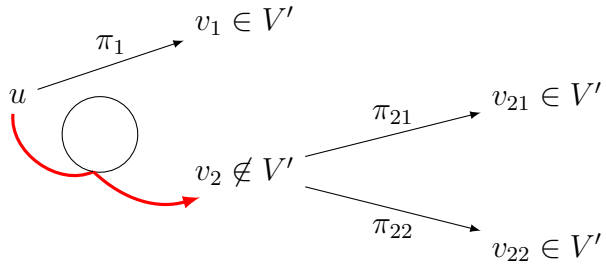


Figure 6.7 – If π_2 contains some cycles, we can remove them

Lemma 6.3

Let $u \in V$. Assume that u is the source of a non-trivial V'-path π_1 and a non-trivial V'-disjoint path π_2 ending in $\mathsf{WD}_G(V')$, such that u is the only vertex present both in π_1 and π_2. Then u is V'-weakly deciding.

Proof.[8] Let v_1 the end of π_1 and v_2 the end of π_2. v_2 is V'-weakly deciding, thus there exist two V'-paths π_{21} ending in v_{21} and π_{22} ending in v_{22} that share no vertex except v_2. This configuration is illustrated in Figure 6.6. Intuitively, u is a V'-weakly deciding vertex, except one of the paths (π_2) ends in $\mathsf{WD}_G(V')$ instead of V'.

First, we can assume without loss of generality that π_2 does not contain any loop, since we can remove them and still have a valid path connecting u and v_2. Figure 6.7 illustrates the removal of the cycles in π_2. We assume in the rest of this proof that every vertex in π_2 is distinct from any vertex in π_2.

Second, we show that we can assume that π_{21} and π_{22} share no vertex with π_2 except v_2. For that, we define u_{21} and u_{22} as the last points on π_{21} and π_{22} respectively that occur in π_2. It is possible that both u_{21} and u_{22} are equal to v_2. Assume without loss of generality that u_{21} does not occur for the first time on π_2

[8] The mechanized version of this proof is available in [Léc18].

after u_{22}. This configuration is represented in Figure 6.8. This means that π_2 can be split into three parts:

- π_{2a} (maybe trivial) between u and u_{21}
- π_{2b} (maybe trivial) between u_{21} and u_{22}
- π_{2c} (maybe trivial) between u_{22} and v_2

π_{21} can be split into two parts:

- π_{21a} (maybe trivial) between v_2 and u_{21}
- π_{21b} between u_{21} and v_{21}, not trivial since $v_{21} \in V'$ while $u_{21} \notin V'$ since u_{21} is in π_2, that has u_{21} as only intersection with π_2

π_{22} can be split into two parts:

- π_{22a} between v_2 and u_{22}
- π_{22b} between u_{22} and v_{22}, not trivial since $v_{22} \in V'$ while $u_{22} \notin V'$ since u_{22} is in π_2, that has u_{22} as only intersection with π_2

In this configuration, we can choose $v'_2 = u_{21}$, $\pi'_2 = \pi_{2a}$, $\pi'_{21} = \pi_{21b}$ and $\pi'_{22} = \pi_{2b}\pi_{22b}$. π'_{21} and π'_{22} are highlighted in Figure 6.8. We can show that v'_2, π'_2, π'_{21} and π'_{22} satisfy the same properties as π_2, π_{21} and π_{22}, with the addition that π'_{21} and π'_{22} share no vertex with π'_2 except u_{21}. Indeed, π'_2 is a V'-disjoint path, since it is the prefix of a V'-disjoint path. π'_{21} and π'_{22} are V'-paths, since they are the suffixes of V'-paths, that have only one vertex in common, u_{21}. Indeed, π_{21b} has only u_{21} as intersection with π_2 thus with π_{2b}, and π_{21b} is disjoint from π_{22b}, because they are suffixes of π_{21} and π_{22}, unless $u_{21} = u_{22} = v_2$. In both cases u_{21} is the only intersection. Moreover, π'_{21} has a single intersection with π_2 and thus with π'_2 since it is a prefix of π_2. Likewise, π'_{22} only intersects π'_2 in u_{22}, because π_2 does not contain any cycle, thus the intersection of π_{2a} and π_{2b} is the vertex u_{21}, and π_{22b} is disjoint π_2 unless $u_{21} = u_{22} = v_2$. We assume in the rest of this proof that π_2 has a single vertex in common with π_{21} and π_{22} respectively, v_2.

Third, we show that we can conclude, but this still requires to consider two cases: the case where π_1 intersects both π_{21} and π_{22} and the case where π_2 is disjoint from at least one of them.
If π_1 intersects both π_{21} and π_{22}, we can assume without loss of generality that it encounters π_{21} first at vertex u_{21}. This case is represented in Figure 6.9. We can split π_1 into π_{1a} and π_{2b}, and π_{21} into π_{21a} and π_{21b}. We can construct two paths from u that share no vertex except u: $\pi_{1a}\pi_{21b}$ and $\pi_2\pi_{22}$. If π_1 does not intersect π_{21} or π_{22}, we can assume without loss of generality that it does not intersect π_{22}.

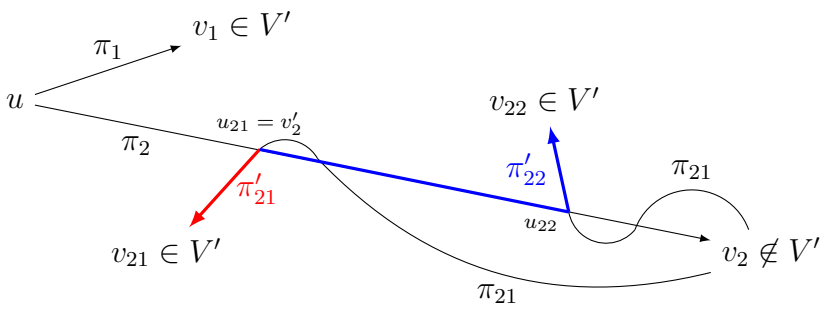


Figure 6.8 – Transformation of the problem when π_2 intersects π_{21} and π_{22}

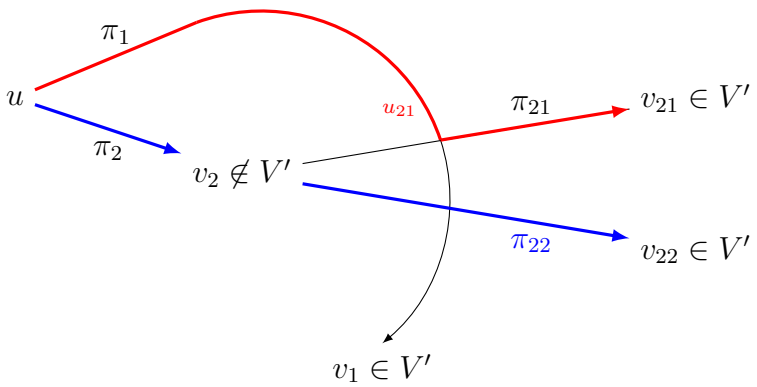


Figure 6.9 – Highlighting of the two V'-paths in the case where π_1 intersects first π_{21} and then π_{22}

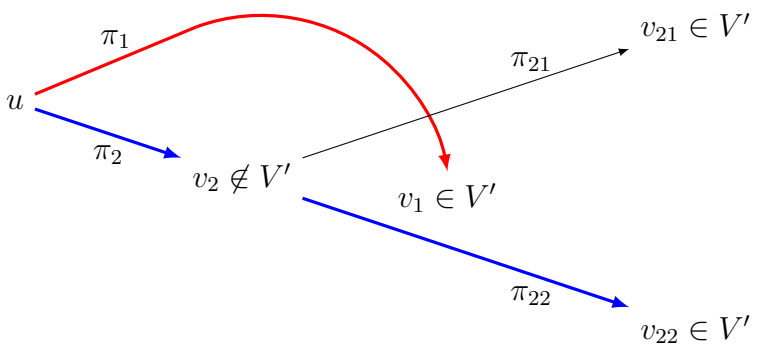


Figure 6.10 – Highlighting of the two V'-paths in the case where π_1 and π_{22} are disjoint

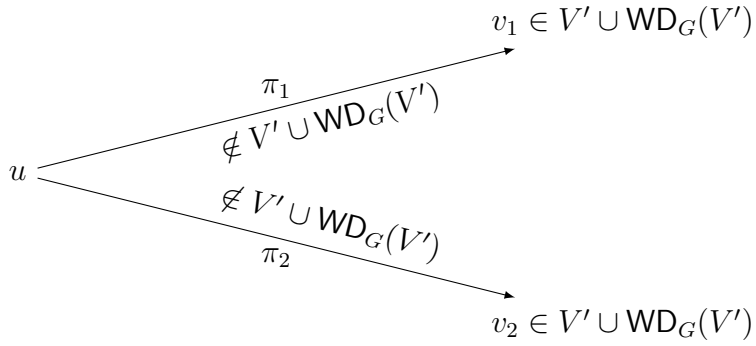


Figure 6.11 – Schematic representation of the configuration of Property 6.4

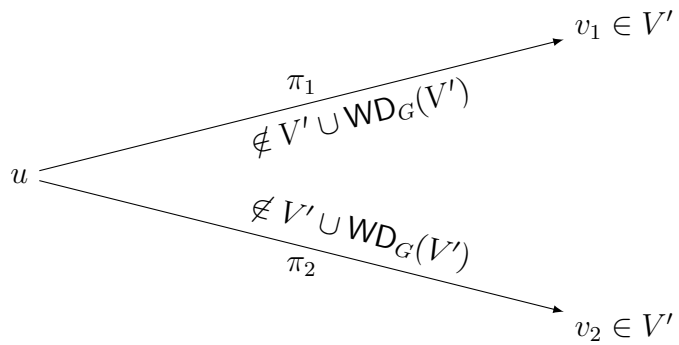


Figure 6.12 – First sub-case of the proof of Property 6.4

In this case, the two V'-paths π_1 and $\pi_2\pi_{22}$ show that u is V'-weakly deciding. This is illustrated in Figure 6.10.

□

Based on Lemma 6.3, we can prove Property 6.4.

Proof of Property 6.4.[9] Let $u \in \mathsf{WD}_G(V' \cup \mathsf{WD}_G(V'))$. There exist two non-trivial $V' \cup \mathsf{WD}_G(V')$-paths π_1 and π_2 ending in v_1 and v_2 respectively that from u that share no vertex except u. This is represented in Figure 6.11.

v_1 and v_2 can both be either in V' or in $\mathsf{WD}_G(V')$. This gives four cases. We show in each case that $u \in \mathsf{WD}_G(V')$.

1. If $v_1 \in V'$ and $v_2 \in V'$ (cf. Figure 6.12), then π_1 and π_2 are not only $V' \cup \mathsf{WD}_G(V')$-paths, but also V'-paths. Therefore, $u \in \mathsf{WD}_G(v')$.

2. If $v_1 \in V'$ and $v_2 \in \mathsf{WD}_G(V')$ (cf. Figure 6.13), we can apply Lemma 6.3 with hypotheses stronger than required. This gives us $u \in \mathsf{WD}_G(V')$.

[9] The mechanized version of this proof is available in [Léc18].

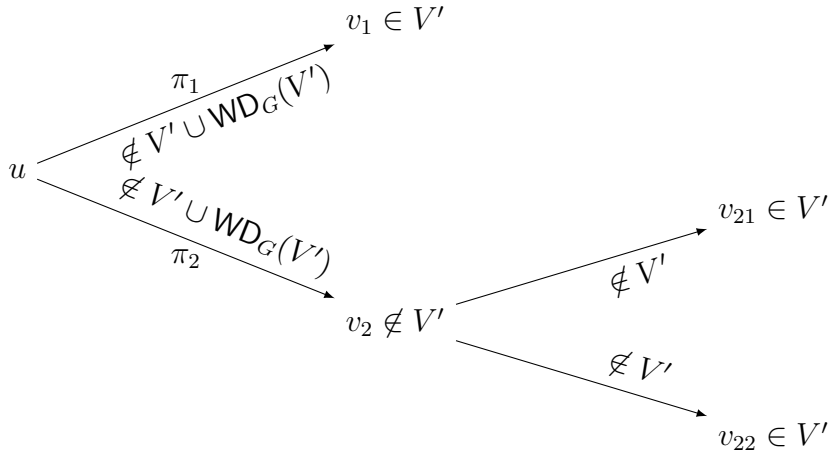


Figure 6.13 – Second sub-case of the proof of Property 6.4

3. If $v_1 \in \mathsf{WD}_G(V')$ and $v_2 \in V'$, we have a configuration symmetrical to the second case and we can apply the same reasoning.

4. If both $v_1 \in \mathsf{WD}_G(V')$ and $v_2 \in \mathsf{WD}_G(V')$ (cf. Figure 6.14), we have two V'-paths π_{11} and π_{12} from v_1, and two V'-paths π_{21} and π_{22} from v_2. Since π_{11} and π_{12} share no vertex except v_1, and $v_1 \neq v_2$, at most one of them can contain v_2. Assume without loss of generality that π_{11} does not contain v_2. If π_2 and π_{11} intersected, the last vertex in π_2 also in π_{11} would satisfy the hypotheses of Lemma 6.3, thus would be in $\mathsf{WD}_G(V')$, which is impossible on π_2, since it is a $V' \cup \mathsf{WD}_G(V')$-path. Thus π_2 and π_{11} do not intersect. By applying again Lemma 6.3 on u and the V'-paths $\pi_1\pi_1 1$ ending in V' and π_2 ending in $\mathsf{WD}_G(V')$, we deduce that $u \in \mathsf{WD}_G(V')$.

In the four cases, we proved that $u \in \mathsf{WD}_G(V')$. Since the choice of u is arbitrary, this means that

$$\mathsf{WD}_G(V' \cup \mathsf{WD}_G(V')) \subseteq \mathsf{WD}_G(V')$$

Since

$$\mathsf{WD}_G(V' \cup \mathsf{WD}_G(V')) \cap (V' \cup \mathsf{WD}_G(V')) = \emptyset$$

this gives the desired result. □

Based on these two lemmas, we can prove that adding to V' the V'-weakly deciding vertices reachable from V' gives a weakly control-closed superset of V'. This is also the smallest weakly control-closed superset of V'.

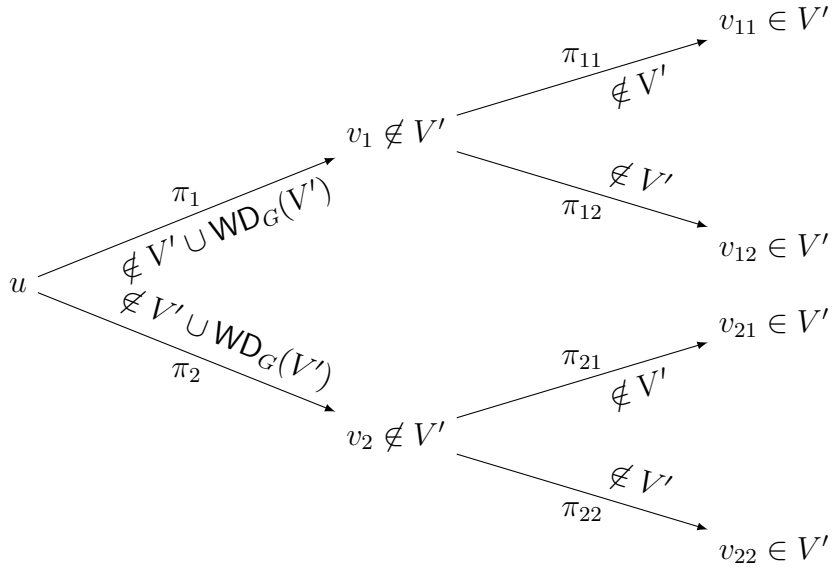


Figure 6.14 – Second sub-case of the proof of Property 6.4

Theorem 6.1: Existence of the weak control-closure

Let $W = \mathsf{WD}_G(V') \cap \mathsf{R}_G(V')$ denote the set of vertices in $\mathsf{WD}_G(V')$ that are reachable from V'. Then $V' \cup W$ is the smallest weakly control-closed set containing V'.

Proof.[10] We prove that $V' \cup W$ is weakly control-closed in G. By Lemma 6.2, it is sufficient to show that there is no element of $\mathsf{WD}_G(V' \cup W)$ reachable from $V' \cup W$. Let u be an element of $\mathsf{WD}_G(V' \cup W)$ reachable from $V' \cup W$. By Property 6.3, since $V' \cup W \subseteq V' \cup \mathsf{WD}_G(V')$,

$$\mathsf{WD}_G(V' \cup W) \subseteq V' \cup \mathsf{WD}_G(V') \cup \mathsf{WD}_G(V' \cup \mathsf{WD}_G(V'))$$

Therefore, by Property 6.4,

$$\mathsf{WD}_G(V' \cup W) \subseteq V' \cup \mathsf{WD}_G(V')$$

Thus $u \in V' \cup \mathsf{WD}_G(V')$. Since u is reachable from $V' \cup W$, and all elements of W are reachable from V', u is reachable from V'. We can deduce that $u \in V' \cup W$. But $u \in \mathsf{WD}_G(V' \cup W)$, which means $u \notin V' \cup W$, which is contradictory. Thus $V' \cup W$ is weakly control-closed in G as expected.

We now prove that $V' \cup W$ is included in any weakly control-closed set containing V'. Let X be a weakly control-closed set containing V' and let u be an

[10] The mechanized version of this proof is available in [Léc18].

element of $V' \cup W$. Either $u \in V'$, and thus $u \in X$, or $u \in W$. Assume that $u \in W$. In particular, $u \in \mathsf{WD}_G(V')$. By Property 6.3, $\mathsf{WD}_G(V') \subseteq X \cup \mathsf{WD}_G(X)$. If u is in X, the proof is done. If $u \in \mathsf{WD}_G(X)$, u is a vertex X-weakly deciding in G, reachable from $V' \cup W$ and thus from X, which is contradictory to X being weakly control-closed in G. □

Definition 6.14: Weak control-closure

We call *weak control-closure* of V', denoted $\mathsf{WCC}_G(V')$, the smallest weakly control-closed set containing V'. We have:

$$\mathsf{WCC}_G(V') = V' \cup \Big(\mathsf{WD}_G(V') \cap \mathsf{R}_G(V')\Big)$$

It follows from this definition that if V' is already weakly control-closed then $\mathsf{WCC}_G(V') = V'$.

Let us illustrate weak control-closure on the running example shown in Figure 6.3. As mentioned above, $\mathsf{WD}_{G_0}(V'_0) = \{u_0, u_2, u_4, u_6\}$, thus by Definition 6.14 and since, as underlined in Section 6.2, each node in G_0 is reachable from V'_0, we have:

$$\mathsf{WCC}_{G_0}(V'_0) = V'_0 \cup \mathsf{WD}_{G_0}(V'_0) = \{u_0, u_1, u_2, u_3, u_4, u_6\}$$

Weak control-closure has several properties that are more natural to express than corresponding ones on V'-weakly deciding vertices.

Property 6.5: Monotonicity of weak control-closure

Let V'_1 and V'_2 be subsets of V, such that $V'_1 \subseteq V'_2$. Then:

$$\mathsf{WCC}_G(V'_1) \subseteq \mathsf{WCC}_G(V'_2)$$

Proof.[11] We know that:

1) $\mathsf{WCC}_G(V'_1) = V'_1 \cup (\mathsf{WD}_G(V'_1) \cap \mathsf{R}_G(V'_1))$

2) $\mathsf{WCC}_G(V'_2) = V'_2 \cup (\mathsf{WD}_G(V'_2) \cap \mathsf{R}_G(V'_2))$

3) $\mathsf{WD}_G(V'_1) \subseteq V'_2 \cup \mathsf{WD}_G(V'_2)$ by Property 6.3

By 3),

$$\mathsf{WD}_G(V'_1) \cap \mathsf{R}_G(V'_1) \subseteq (V'_2 \cup \mathsf{WD}_G(V'_2)) \cap \mathsf{R}_G(V'_1)$$

[11] The mechanized version of this proof is available in [Léc18].

Since $V_1' \subseteq V_2'$, $\mathsf{R}_G(V_1') \subseteq \mathsf{R}_G(V_2')$, and thus:

$$\mathsf{WD}_G(V_1') \cap \mathsf{R}_G(V_1') \subseteq (V_2' \cup \mathsf{WD}_G(V_2')) \cap \mathsf{R}_G(V_2')$$

Whence, since $V_2' \subseteq \mathsf{R}_G(V_2')$,

$$\mathsf{WD}_G(V_1') \cap \mathsf{R}_G(V_1') \subseteq V_2' \cup \mathsf{WD}_G(V_2') \cap \mathsf{R}_G(V_2')$$

Finally, using $V_1' \subseteq V_2'$,

$$V_1' \cup \mathsf{WD}_G(V_1') \cap \mathsf{R}_G(V_1') \subseteq V_2' \cup \mathsf{WD}_G(V_2') \cap \mathsf{R}_G(V_2')$$

i.e. $\mathsf{WCC}_G(V_1') \subseteq \mathsf{WCC}_G(V_2')$. □

Property 6.6: Idempotence of weak control-closure

$\mathsf{WCC}_G(\mathsf{WCC}_G(V')) = \mathsf{WCC}_G(V')$

Proof.[12] $\mathsf{WCC}_G(V')$ is already weakly control-closed, thus by Definition 6.14, its weak control-closure is itself. □

6.4 Link with Classic Control Dependence

The objective of this section is to illustrate on program p shown in Figure 4.2 that this new notion of weak control-closed set subsumes being closed under a classic form of control dependence. For that we will show on the slicing criterion of Figure 4.4 that interlacing computations of weak control-closure with classic data dependencies gives the same slice set as classic static backward slicing.

To apply Danicic et al.'s definitions, we consider the CFG of p that is represented in Figure 6.1.

To make the comparison, we need to take as slicing criterion the equivalent to $\{8\}$, the slicing criterion of Figure 4.4. Danicic et al. [DBH+11, Section 7] showed that weak control-closure and closure under a classic form of control dependence are comparable when `enter` and `exit` nodes are in the slicing criterion. Thus, we choose as slicing criterion: $V_1' = \{\texttt{enter}, u_8, \texttt{exit}\}$.

Since `enter` can reach every vertex in the graph, to build the weak control-closure of V_1', it is sufficient to add to V_1' every V_1'-weakly deciding vertex. u_6 is V_1'-weakly deciding, since there are two non-trivial V'-paths $u_6, u_7, \texttt{exit}$ and u_6, u_8 that share no vertex except u_6. u_6 and its associated V_1'-paths are highlighted in

[12]The mechanized version of this proof is available in [Léc18].

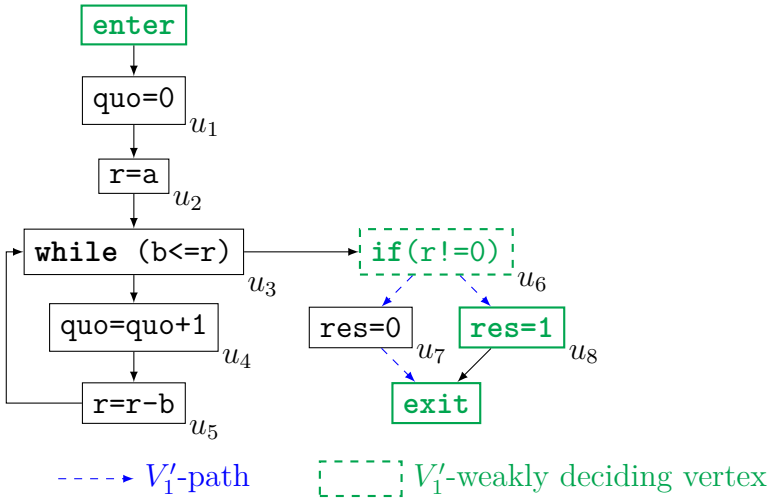


Figure 6.15 – Control flow graph G of program p of Figure 4.2, with slicing criterion $V_1' = \{\texttt{enter}, u_8, \texttt{exit}\}$

Figure 6.15. This is the only V_1'-weakly deciding vertex, thus the first step returns $V_{11}' = \mathsf{WCC}_G(V_1') = \{\texttt{enter}, u_6, u_8, \texttt{exit}\}$.

Since `r` is used at u_6 and may be defined for the last time at u_5 and u_2, data dependencies add vertices u_5 and u_2 to the slice set, resulting in the set $V_{12}' = \{\texttt{enter}, u_2, u_5, u_6, u_8, \texttt{exit}\}$, represented in Figure 6.16.

The next step consists in computing the weak control-closure of V_{12}'. u_3 is V_{12}'-weakly deciding, the two V_{12}'-paths being u_3, u_4, u_5 and u_3, u_6 highlighted in Figure 6.16. This is the only V_{12}'-weakly deciding vertex. The weak control-closure of V_{12}' is therefore $V_{13}' = \{\texttt{enter}, u_2, u_3, u_5, u_6, u_8, \texttt{exit}\}$. Data dependence does not add any additional vertex. The resulting slice set is thus $V_{13}' = \{\texttt{enter}, u_2, u_3, u_5, u_6, u_8, \texttt{exit}\}$, which corresponds as expected to the slice set $\{2, 3, 5, 6, 8\}$ computed in Figure 4.4.

6.5 Danicic's Algorithm for Computing Weak Control-Closure

Definition 6.14 gives an explicit definition of weak control-closure. One possible algorithm could be to apply the definition literally and test each vertex in the graph to check whether it is V'-weakly deciding. If yes and if it is additionally reachable from V', then we add it to V'. By definition, the resulting set is the weak control-closure of V' in G.

But this is not the path followed by Danicic et al [DBH+11]. Instead, they

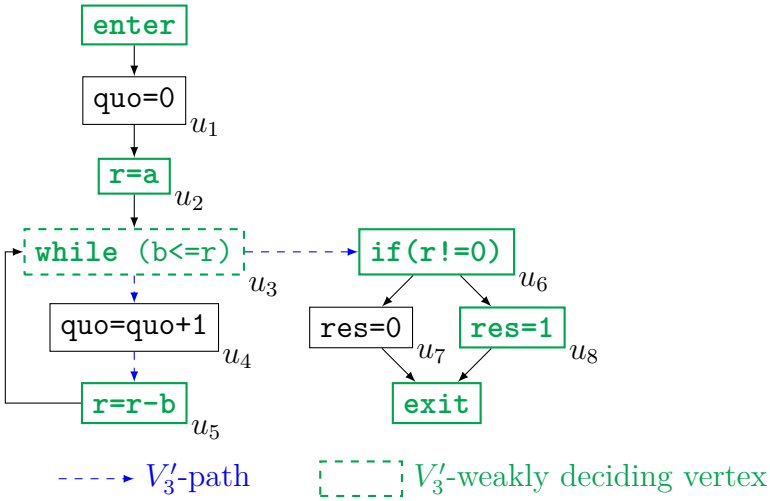


Figure 6.16 – Control flow graph G of program p of Figure 4.2, with slicing criterion $V'_{12} = \{\texttt{enter}, u_2, u_5, u_6, u_8, \texttt{exit}\}$

described an iterative algorithm relying on a sufficient condition for a node to be V'-weakly deciding.

First, we introduce this sufficient condition in Section 6.5.1 and study its properties. Then, we describe in Section 6.5.2 the algorithm designed by Danicic et al., from now on referred to as Danicic's algorithm. The correctness of Danicic's algorithm is completely guaranteed by the properties proved in Section 6.5.1.

6.5.1 Critical Edge

This section introduces V'-*critical edges*, which are edges in G whose origins are V'-weakly deciding. Such edges are introduced by Danicic, but were not given a name. We chose the name "critical edge". Showing that a vertex is the origin of a V'-critical edge is thus a sufficient condition for proving it is V'-weakly deciding. However, this is not a necessary condition. Every V'-weakly deciding vertex in G is not the start node of a V'-critical edge. But critical edges have a second interesting property. When there is no V'-critical edge in G whose source is reachable from V', then there is no V'-weakly deciding vertex reachable from V' either, and thus V' is weakly control-closed in G.

Definition 6.15: Critical edge

An edge $(u, v) \in E$ is called V'-*critical* if:

(1) $|\,\mathsf{obs}_G(u, V')| \geq 2$;

(2) $|\,\mathsf{obs}_G(v, V')| = 1$;

(3) u is reachable from V' in G.

In our example graph G_0 shown in Figure 6.4 with its vertices annotated with their observable vertices in V_0', (u_0, u_1) is a V_0'-critical edge. Indeed, we have $|\,\mathsf{obs}_{G_0}(u_0, V_0')| = 2$ and $|\,\mathsf{obs}_{G_0}(u_1, V_0')| = 1$. Likewise, (u_2, u_3) and (u_4, u_3) are V_0'-critical edges. It should be noted that u_6 is not the source of a V'-critical edge, since its two children u_4 and u_0 both have two observable vertices in V_0', while being V_0'-weakly deciding as shown in Section 6.3, illustrating that finding a critical edge is not a necessary condition for finding a weakly deciding vertex.

We now prove that V'-critical edges are indeed good objects when looking for V'-weakly deciding vertices.

Lemma 6.4

If V' is not weakly control-closed in G, then there exists a V'-critical edge (u, v) in G. Moreover, if (u, v) is such a V'-critical edge, then u is a V'-weakly deciding vertex reachable from V', and thus $u \in \mathsf{WCC}_G(V')$.

Proof.[13] Let x be a vertex in $\mathsf{WD}_G(V')$ reachable from V'. By Definition 6.12, there exist two V'-paths π_1 and π_2 ending in x_1 and x_2 respectively that share no vertex except x. At the beginning of π_1, we have $\mathsf{obs}_G(x, V') \supseteq \{x_1, x_2\}$ and thus $|\,\mathsf{obs}_G(x, V')| \geq 2$. At the end of π_1, we have $\mathsf{obs}_G(x_1, V') = \{x_1\}$ and thus $|\,\mathsf{obs}_G(x_1, V')| = 1$. Let u be the last vertex on π_1 with at least two observable nodes in V' and v be its successor on π_1. Then, by definition of u, $|\,\mathsf{obs}_G(u, V')| \geq 2$ and $|\,\mathsf{obs}_G(v, V')| < 2$. Since $\mathsf{obs}_G(v, V') \supseteq \{x_1\}$, $|\,\mathsf{obs}_G(v, V')| \geq 1$ and therefore $|\,\mathsf{obs}_G(v, V')| = 1$. Moreover, u is reachable from x which is itself reachable from V'. u is therefore reachable from V'. By Definition 6.15, (u, v) is a V'-critical edge.

Assume there exists a V'-critical edge (u, v). We know, by Definition 6.15, that $|\,\mathsf{obs}_G(v, V')| = 1$. Let w be the observable vertex in V' from v and π be a V'-path from v to w. Since $(u, v) \in E$, w is also an observable vertex in V' from u. But $|\,\mathsf{obs}_G(u, V')| \geq 2$, thus u has another observable vertex $w_2 \neq w$ in V' which is

[13] The mechanized version of this proof is available in [Léc18].

not an observable vertex from v. Let π_2 be a V'-path from u to w_2. π and π_2 are two V'-paths from u that are non-trivial, otherwise $u \in V'$ which contradicts $|\,\mathsf{obs}_G(u, V')| \geq 2$. If π and π_2 intersected at another vertex than u, this would mean that $w_2 \in \mathsf{obs}_G(v, V')$ which would contradict the definition of w_2. Thus, π and π_2 have only u in common. By Definition 6.12, u is V'-weakly deciding. Moreover, (u, v) being a V'-critical edge also means that u is reachable from V'. This leads to $u \in \mathsf{WCC}_G(V')$. □

We can note that the key argument of the proof above is the fact that u has at least one observable vertex in V' that is not an observable vertex from v in V'. v also needs to have at least one observable vertex in V'. We could thus weaken the definition of V'-critical edge and just require that

$$1 \leq |\,\mathsf{obs}_G(v, V')| < |\,\mathsf{obs}_G(u, V')|$$

6.5.2 Definition and Proof of Correctness of Danicic's Algorithm

Danicic's algorithm is entirely based on Lemma 6.4. It consists in detecting critical edges and adding their sources. Indeed, by Lemma 6.4, it is clear that such vertices belong to the weak control-closure. But it was also shown in Section 6.5.1 that being the source of a V'-critical edge is not a necessary condition for being V'-weakly deciding and reachable from V'. If applied only on V', this procedure will thus not find all V'-weakly deciding vertices that are reachable from V'.

The idea of the algorithm is to proceed iteratively. It manipulates an intermediate set W initialized to V' that grows during each iteration until it is equal to the weak control-closure of V'. More precisely, an iteration consists in looking for a W-critical edge and, if one is found, adding its source to W. When there is not any W-critical edge any more, the algorithm stops and returns W. Danicic's algorithm is presented as Algorithm 6.1.

It is of primary importance that the algorithm looks for W-critical edges instead of V'-critical edges. Indeed, as recalled just above, every vertex that needs to be added to build the weak control-closure of V', i.e. every vertex in $\mathsf{WCC}_G(V') \setminus V'$, is not the source of a V'-critical edge. This means that during the first iteration, when $W = V'$, it is not possible either to find every vertex in $\mathsf{WCC}_G(V') \setminus V'$. But the correctness of the algorithm relies on the fact that every vertex in $\mathsf{WCC}_G(V') \setminus V'$ becomes the source of a W-critical edge after some iterations.

The correctness of Danicic's algorithm relies on the invariant $V' \subseteq W \subseteq \mathsf{WCC}_G(V')$. At the beginning of the algorithm, $W = V'$ and thus $V' \subseteq W \subseteq \mathsf{WCC}_G(V')$. Assume that at the beginning of an iteration $V' \subseteq W \subseteq \mathsf{WCC}_G(V')$

Input: $G = (V, E)$ a directed graph
$V' \subseteq V$
Output: $W \subseteq V$ the weak control-closure of V'
Ensures: $W = \mathsf{WCC}_G(V')$

```
1 begin
2     W ← V'
3     while there exists a W-critical edge in E do
4         choose such a W-critical edge (u, v)
5         W ← W ∪ {u}
6     end
7     return W
8 end
```

Algorithm 6.1: Danicic's algorithm for computing weak control-closure

and there exists a W-critical edge (u, v). By Lemma 6.4, u is a W-weakly deciding vertex reachable from W, i.e. $u \in \mathsf{WCC}_G(W)$. Since $V' \subseteq W$, we have trivially $V' \subseteq W \cup \{u\}$. By Property 6.5, WCC_G is monotonic and thus from $W \subseteq \mathsf{WCC}_G(V')$, we can deduce $\mathsf{WCC}_G(W) \subseteq \mathsf{WCC}_G(\mathsf{WCC}_G(V'))$. By Property 6.6, WCC_G is idempotent, so we can conclude $\mathsf{WCC}_G(W) \subseteq \mathsf{WCC}_G(V')$. Combining these results, we get $V' \subseteq W \cup \{u\} \subseteq \mathsf{WCC}_G(V')$, which proves that the invariant is preserved after the iteration.

At the end of the algorithm, $V' \subseteq W \subseteq \mathsf{WCC}_G(V')$. By the monotonicity and the idempotence of WCC_G (Properties 6.5 and 6.6), this gives $\mathsf{WCC}_G(W) = \mathsf{WCC}_G(V')$. Moreover, there does not exist any W-critical edge. By Lemma 6.4, W is weakly control-closed in G, i.e. $W = \mathsf{WCC}_G(W)$. At the end of the algorithm, we have $W = \mathsf{WCC}_G(V')$ as expected.

As for the termination of Danicic's algorithm, W strictly increases as long as W-critical edges are found and is upper-bounded by $\mathsf{WCC}_G(V')$.

At a given iteration, to find W-critical edges, Danicic's algorithm computes the set of observable vertices of multiple vertices in G. Since W is modified afterwards, these computations are no longer valid and must be redone. This observation leads to two possible optimizations.

First, it is possible to add multiple vertices during each iteration. This will reduce the number of iterations and thus avoid recomputing the set of observable vertices too often. This change is correct, since the proof of correctness of the algorithm relies on the fact that the singleton added to W is a subset of $\mathsf{WCC}_G(W)$. If we add multiple vertices, we also add to W a subset of $\mathsf{WCC}_G(W)$. Thus, at the end, the algorithm still returns the weak control-closure of V'.

Second, it is possible to weaken the definition of critical edge as explained in

Section 6.5.1. Combined with the first optimization, this will further reduce the number of iterations.

To illustrate Danicic's algorithm on our running example graph G_0 shown in Figure 6.3, we apply the first optimization that we have just described for the sake of concision, i.e. we have the possibility to add several vertices in one iteration. The different steps are given in Figure 6.17. The algorithm starts with $W = V_0' = \{u_1, u_3\}$ (Figure 6.17a). The first step consists in computing the set of observable vertices of each vertex in W (Figure 6.17b). This reveals three W-critical edges: (u_0, u_1), (u_2, u_3) and (u_4, u_3). We add u_0, u_2 and u_4 to W, and throws away the annotations that are no longer valid (Figure 6.17c). Again, we compute the set of observable vertices of each vertex in W (Figure 6.17d). u_6 is the origin of two W-critical edges: (u_6, u_0) and (u_6, u_4). We add u_6 to W and again remove the annotations (Figure 6.17e). Once more, we compute the set of observable vertices of each vertex in W (Figure 6.17f). This time, no W-critical edge is detected. This ends the algorithm. The computed closure is $W = \{u_0, u_1, u_2, u_3, u_4, u_6\}$. It is indeed equal to the weak control-closure of V_0' computed from Definition 6.14 in Section 6.3.

In terms of complexity, Danicic et al. [DBH+11] show the complexity of their algorithm $O(|V|^3)$, under the assumption that the degree of each vertex is at most 2 (and thus that $O(|V|) = O(|E|)$). Indeed, the main loop of Algorithm 6.1 is run at most $O(|V|)$ times, and each loop body computes obs in $O(|V|)$ for at most $O(|V|)$ edges.

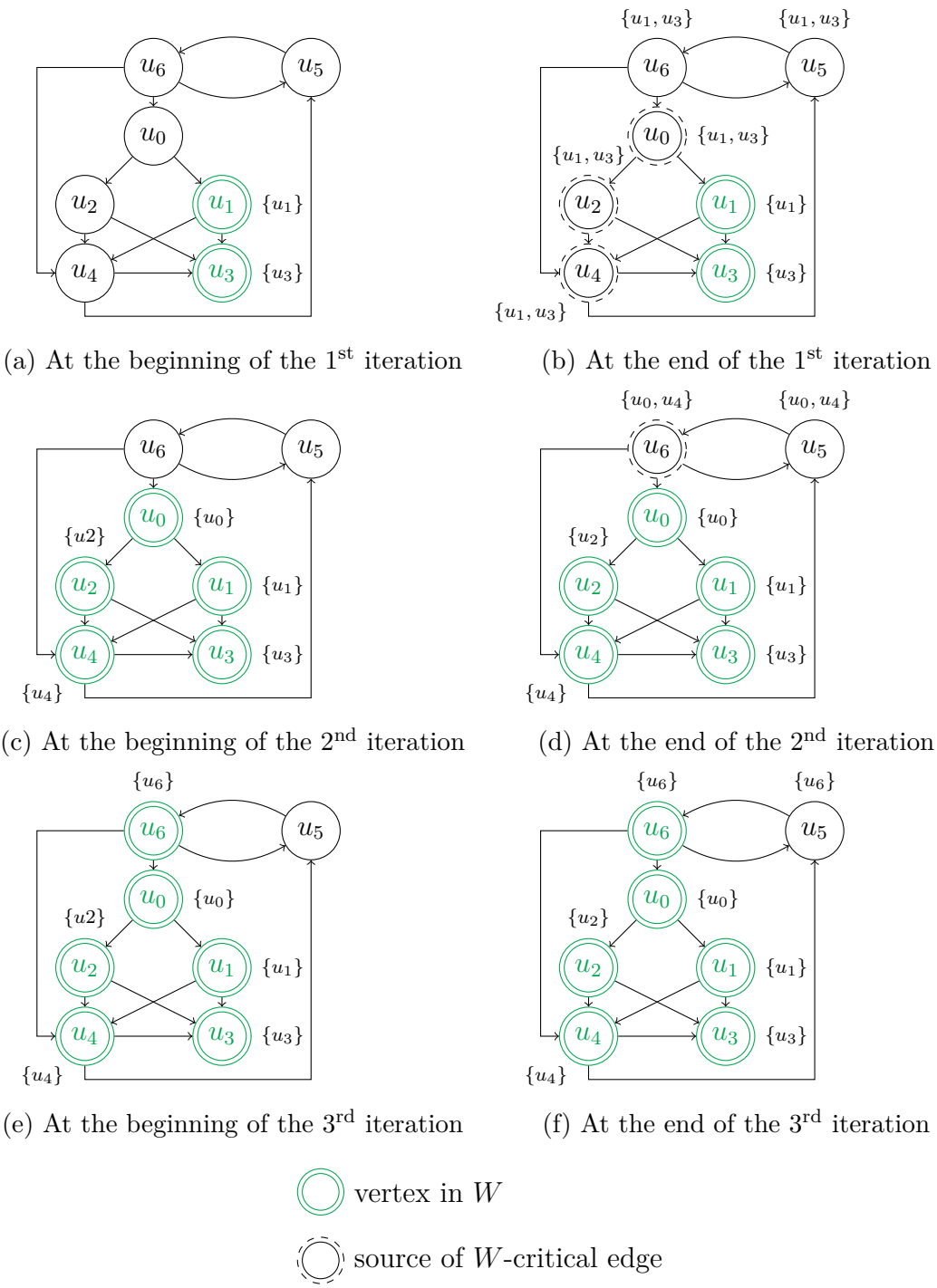
6.6 Remarks about the Coq Formalization

The concepts presented in this chapter are taken from [DBH+11]. The contribution of this chapter is therefore not the concepts themselves, but their formalization in Coq. This formalization [Léc18] contains 6 600 lines of Coq code (2 000 lines of specification, 4 600 lines of proof).

The previous sections gave a mathematical presentation of the Coq formalization. We highlight some major points of the Coq development in Section 6.6.1 and make some remarks about it in the next subsections.

6.6.1 Coq Formalization

The Coq development takes advantage of Coq's module system to organize the results into multiple refinements. One advantage is that it clearly separates the theoretical results from the algorithm.



(a) At the beginning of the 1st iteration

(b) At the end of the 1st iteration

(c) At the beginning of the 2nd iteration

(d) At the end of the 2nd iteration

(e) At the beginning of the 3rd iteration

(f) At the end of the 3rd iteration

Figure 6.17 – Optimized Danicic's algorithm applied on G_0 (cf. Figure 6.3) with $V_0' = \{u_1, u_3\}$

```
Module Type ABSTRACT_GRAPH (V : UsualEq) (L : LABEL).
    Parameter Graph : Type.
    Parameter Rel : Graph → V.t → Label → V.t → Prop.
    Parameter Support : Graph → Ensemble V.t.
    Axiom Support_spec : forall g u l v,
        Rel g u l v → Support g u ∧ Support g v.
End ABSTRACT_GRAPH.
```

Figure 6.18 – Abstract specification of graphs

```
Module Type ABSTRACT_FINITE_GRAPH (V : UsualEq) (L: LABEL).
    Include ABSTRACT_GRAPH V L.

    Axiom Support_finite : forall g, Finite (Support g).
    Axiom Rel_finite : forall g u,
        Finite (fun v ⇒ exists l, Rel g u l v).
End ABSTRACT_FINITE_GRAPH.
```

Figure 6.19 – Abstract specification of finite graphs

Let us first present the two abstract signatures of graphs that we use. They are shown in a slightly simplified form in Figure 6.18 and Figure 6.19 respectively.

The most abstract signature is `ABSTRACT_GRAPH`. The type of graph, `Graph`, is completely abstract. Two predicates are provided: `Support` that returns the set of nodes in the graph and `Rel` that models the edge relation. Note that relations in `Rel` are labeled. These labels are used by Danicic et al. [DBH+11] when characterizing the semantics of control dependence. Labels are also present in the Coq development, but since we do not formalize this part of Danicic et al.'s theory (see Section 6.6.2), they are not really used. Axiom `Support_spec` expresses the consistency between `Support` and `Rel`. It specifies that the end nodes of an edge are necessarily vertices of the graph.

Signature `ABSTRACT_FINITE_GRAPH` adds two finiteness hypotheses to signature `ABSTRACT_GRAPH`. `Support_finite` states that the graph contains a finite number of vertices, while `Rel_finite` states that each vertex has a finite number of outgoing edges.

We can now present the four functors structuring the Coq formalization. They are presented in a simplified version in Figure 6.20, ordered from the most theoretical to the least theoretical.

```
Module AbstractGraphProperties (V : UsualEq) (L : LABEL)
    (G : ABSTRACT_GRAPH V L).

Module AbstractGraphPropertiesDec
  (V : UsualDecidableType) (L : LABEL) (G : ABSTRACT_GRAPH V C).
    Include AbstractGraphProperties V L G.

Module AbstractGraphPropertiesDecFinite (V : UsualDecidableType)
  (L : LABEL) (G : ABSTRACT_FINITE_GRAPH V C).
    Include AbstractGraphPropertiesDec V L G.

Module preCFG (Vertex : UsualOrderedType) (Label : Labels)
    (DG : Constructive_Directed_Graph Vertex Label).
    Module V <: UsualDecidableType.
    Module L <: LABEL.
    Module G <: ABSTRACT_FINITE_GRAPH V C.

    Module Graph := AbstractGraphPropertiesDecFinite V C G.
```

Figure 6.20 – Hierarchy of functors

```
Definition Weakly_control_closed G V := forall u,
    Reachable_from G V u → Weakly_committing G V u.
```

Figure 6.21 – Definition of weakly control-closed set in Coq

```
Lemma lemma53 : forall g V,
    Weakly_deciding g (Union V (Weakly_deciding g V))
    == Empty_set.
```

Figure 6.22 – Formulation of Property 6.4 in Coq ("53" refers to [DBH+11, Lemma 53])

`AbstractGraphProperties`. Functor `AbstractGraphProperties` defines most of the basic definitions and some results. Graphs and sets are defined in an abstract way using predicates in `Prop`. In particular, signature `ABSTRACT_GRAPH` is used, which means that no hypothesis is made on the finiteness of the graph. For instance, weakly control-closed sets (cf. Definition 6.11) are defined in this functor. The exact definition is given in Figure 6.21.

`AbstractGraphPropertiesDec`. Functor `AbstractGraphPropertiesDec` makes the additional hypothesis that the equality over vertices is decidable, which means that functions are allowed to compare vertices. It inherits all the definitions and results of `AbstractGraphProperties`. It especially contains the proof of Property 6.4. Its formulation in Coq is given in Figure 6.22.

`AbstractGraphPropertiesDecFinite`. The next stage in the hierarchy is the functor `AbstractGraphPropertiesDecFinite` that further assumes that graphs are finite, since it uses `ABSTRACT_FINITE_GRAPH` and no longer `ABSTRACT_GRAPH`. It inherits all the previous definitions and results of `AbstractGraphPropertiesDec`. In particular, it contains the proof of Theorem 6.1, whose statement is shown as Figure 6.23. `Minimal_Weakly_control_closed` is the explicit definition of the weak control-closure (cf. Definition 6.14): the union of the initial set and the weakly-deciding nodes reachable from it. Theorem `th54_explicit` states that `Minimal_Weakly_control_closed` is the minimum of the weakly control-closed supersets, i.e. it contains the initial set, it is weakly control-closed and it is included in any weakly control-closed set containing the initial set. The hypotheses about decidability of membership (of the form "`forall u, X u ∨ ~ X u`", meaning that for every vertex `u`, either `u` is in `X` or `u` is not in `X`) could be removed if the considered sets were known to be finite. Indeed, in that case, membership is decidable since it is sufficient to enumerate all the elements of the set to decide whether a vertex

```
Definition Minimal_Weakly_control_closed g V :=
  let W := fun u ⇒ Weakly_deciding g V u ∧ Reachable_from g V u in
  Union V W.

Theorem th54_explicit : forall g V (V_In_dec : forall u, V u ∨ ~ V u),
    Minimum (fun X ⇒ Included V X
                     ∧ (forall u, X u ∨ ~ X u)
                     ∧ Weakly_control_closed g X
            ) (Minimal_Weakly_control_closed g V).
```

Figure 6.23 – Formulation of Theorem 6.1 in Coq ("54" refers to [DBH+11, Theorem 54])

```
Lemma real_algo61_correct : forall g U,
    Ensemble_of_set (real_algo61 g U)
    == Minimal_Weakly_control_closed g (Ensemble_of_set U).
```

Figure 6.24 – Formulation of the correctness of Danicic's algorithm in Coq ("61" refers to [DBH+11, Algorithm 61])

belongs to it. For instance, we could restrict the theorem to subsets in the graph, that would be finite since the graph is finite. This is not done here.

preCFG. Functor `preCFG` takes as inputs concrete representations of vertices (`Vertex`), labels (`Label`) and directed graphs (`DG`) defined as maps of vertices. It then instantiate modules `V`, `L` and `G` to make the link between the concrete representations and the abstract ones. Using these modules, it instantiates functor `AbstractGraphPropertiesDecFinite` to get all the previous results. Using the concrete representation of graphs, it defines Danicic's algorithm (cf. Algorithm 6.1) using the first optimization described in Section 6.5) and proves it correct. Figure 6.24 shows the main result of `preCFG`: the set returned by the algorithm (called `real_algo61`) is equal to the weak control-closure.

To apply Danicic's algorithm in Coq, it is thus sufficient to instantiate functor `preCFG` by specifying the exact types of labels, vertices and graphs. This results in a function `real_algo61` that can be called on concrete graphs. It can also be extracted into OCaml. This gives a certified implementation of Danicic's algorithm in OCaml. To interface with other OCaml code, one can write a small wrapper as mentioned in 5.5.1.

6.6.2 No Use of Coinductive Types

As for the WHILE case (cf. Section 5.5.4), projections are a problem. Indeed, the paths in the graphs are modeled using `CoInductive`. It is not clear how to define projection, either with `CoInductive` or `CoFixpoint`, due to the guard condition that prevents an infinite number of removals. This partly explains why we did not model the part of Danicic et al.'s theory about giving semantics to weak control dependence.

6.6.3 No Generic Graph Library

There seems to be no generic Coq library manipulating graphs à la OCamlgraph [CFS07] in OCaml. We used a prototype library presented in [DERV15].

6.6.4 Advantage of a Proof Assistant for Graph Theoretic Proofs

Several proofs had to manipulate paths in a non-trivial way, especially the proofs manipulating weakly deciding vertices. This is particularly true for the proof of Lemma 6.3, which has to consider a lot of configurations. As usually in this kind of situations, a proof assistant is very helpful to ensure that every corner case is indeed taken into account and exclude any possible error.

6.6.5 Termination of Danicic's Algorithm

The algorithm iterates until no node can be added. Termination is not structural. Therefore, Coq cannot prove it alone. As usual, a difficulty was to find which Coq construct (among `Function`, `Program Fixpoint`, etc.) and which well-founded order were the best ones.

We chose to use `Function` which was powerful enough for our purpose. The chosen well-founded order $<$ is the strict subset inclusion of the complements in the total set V of vertices of the graph:

$$W_1 < W_2 \iff V \setminus W_1 \subset V \setminus W_2$$

Indeed, in Danicic's algorithm (cf. Algorithm 6.1), a new iteration is done only if a vertex is added to the working set W, i.e. only if its complement strictly decreases.

This chapter presented the formalization in Coq of a fragment of a generalization of control dependence for arbitrary finite directed graphs due to Danicic et

al. [DBH+11].

The Coq formalization includes:

- the definition of weak control-closed sets, generalizing sets closed under control dependence;
- the proof of existence, for any arbitrary subset, of the smallest superset weakly control-closed, called the weak control-closure of that subset;
- the algorithm computing weak control-closure proposed by Danicic et al.;
- the proof of correctness of this algorithm.

The Coq formalization of the algorithm can be extracted into OCaml, giving a certified implementation of Danicic's algorithm.

However, as mentioned in Section 6.5.2, Danicic's algorithm is cubic. We have already proposed two optimizations (cf. Section 6.5.2, but there seems to be some further room for improvement, as acknowledged by Danicic et al. themselves: "It is believed that better than $O(|V|^3)$ worst-case time complexity algorithms may exist". In the next chapter, we propose a new algorithm computing weak control-closure that optimizes Danicic's algorithm. Then, we prove it correct in Chapter 8.

Chapter 7

Design of a New Algorithm Computing Weak Control-Closure

Danicic's algorithm, presented in Section 6.5.2, is iterative. This means that it applies repeatedly the same instructions until some condition becomes false. This kind of algorithm can be improved in two ways:

- improving the instructions run at each iteration, so that each iteration does more work in less time;
- improves the transfer of information from an iteration to the following ones, so that these following iterations can take advantage of that information and do less computation.

The two optimizations proposed to enhance Danicic's algorithm in Section 6.5.2, relaxing the critical edge definition and detecting as many critical edges as possible during each iteration, are of the first kind, since they allow to detect more weakly deciding vertices at each iteration. They are particularly interesting (see Chapter 9), because Danicic's algorithm does not share any information between iterations. Indeed, as mentioned in Section 6.5.2 and illustrated by Figure 6.17, at the end of an iteration, all the information about observable sets is no longer valid since the set with respect to which this information was computed is modified. Starting a new iteration means computing again some observable information from scratch, it is thus costly. Both optimizations reduce the number of iterations, thus reduce the number of times this information needs to be computed. They would be less interesting if starting a new iteration were not so costly.

In this chapter, we address the second kind of optimization. We label each node in the graph with some information about its observable set, that is not removed at the end of an iteration and can thus be used by the following iterations.

This chapter is organized as follows. Section 7.1 presents the general idea of the algorithm. Next, Section 7.2 gives a precise but informal description of the algorithm and illustrates it on the running example of Figure 6.3. Then, Section 7.3 gives the formal definition of the algorithm and describes its three functions.

7.1 General Idea of the Optimization

In this section, $G = (V, E)$ denotes a finite directed graph with finite set of vertices V and set of edges $E \subseteq V \times V$, and V' denotes a subset of V.

We introduce our algorithm by highlighting its similarities and differences with Danicic's algorithm.

Contrary to Danicic's algorithm, the new algorithm does not compute directly the weak control-closure of the initial set V', but instead the set $V' \cup \mathsf{WD}_G(V')$. To get the weak control-closure $\mathsf{WCC}_G(V')$, we can note that

$$\mathsf{WCC}_G(V') = V' \cup (\mathsf{WD}_G(V') \cap \mathsf{R}_G(V')) = (V' \cup \mathsf{WD}_G(V')) \cap \mathsf{R}_G(V')$$

since every node in V' is reachable from V'. It is thus sufficient to filter $V' \cup \mathsf{WD}_G(V')$ at the end of the algorithm to keep only vertices reachable from V'. This choice to ignore the reachability tests during the algorithm and make them afterwards helps separate concerns and focus on the core part of the algorithm. Moreover, in the case where we want the same results for slicing based on Danicic et al.'s weak control dependence and for slicing based on classic control dependence, we need to add to V' an `enter` node that reaches every other node in G, as described in Section 6.4. In this case, all the nodes in G are reachable from V', thus the question whether only reachable vertices are considered makes no difference.

Although the new algorithm and Danicic's one do not compute the exact same results, they share the same structure. Our algorithm is also an iterative algorithm which manipulates a set of vertices W equal to V' initially, that grows during the execution of the algorithm, and is equal to the result at the end, i.e. $V' \cup \mathsf{WD}_G(V')$ for our algorithm. Like Danicic's algorithm, during an iteration, our algorithm detects some kind of W-critical edges (u, v) and adds u to W. The main difference is the way our algorithm detects these edges based on the labeling of the vertices in the graph.

To introduce the labeling in more detail, we need to make precise the kind of W-critical edge our algorithm is based on. Danicic's algorithm proposes to detect edges (u, v) such that (cf. Definition 6.15):

(1) $|\,\mathsf{obs}_G(u, W)| \geq 2$;

(2) $|\,\mathsf{obs}_G(v, W)| = 1$;

(3) u is reachable from W in G.

Condition (3) is needed to consider only vertices reachable from V', it is not useful for our algorithm since we do not return the closure but $V' \cup \mathsf{WD}_G(V')$. Conditions (1) and (2) together are sufficient to make u a W-weakly deciding vertex (cf. the proof of Lemma 6.4). But, as noted at the end of Section 6.5.1, the key point in this definition is that u should have at least one observable vertex in W which is not observable from v. In Section 6.5.1, this point is used to justify the following weakened definition of W-critical edge:

$$1 \leq |\,\mathsf{obs}_G(v, W)| < |\,\mathsf{obs}_G(u, W)|$$

Whether the initial or the weakened definition is used, a direct approach requires to compute the complete set of observable vertices for the considered nodes. But the key point of Section 6.5.1 actually gives a simpler sufficient condition to find W-weakly deciding vertices. To prove that a vertex u is W-weakly deciding, we can just find some u' in W and some child v of u such that:

- v can reach W,
- u' is observable from u in W, and
- u' is not observable from v in W.

It is thus not needed to compute the whole set of observable nodes, exhibiting such an observable vertex u' is enough.

Given a vertex u, proving that a vertex u' in W is observable from u in W is simple. It is sufficient to exhibit a W-path from u to u'. Such a path can be revealed by traversing backwards the graph from u' and stopping when encountering W. All the nodes traversed have u' as observable vertex in W.

Proving that a vertex u' in W is not observable from v in W may seem more complicated. But actually, the same backward traversal also addresses this question, since not only all the nodes considered during the traversal have u' as observable vertex in W, but they are exactly *the* nodes that have u' as observable vertex in W. This means that the other nodes do not have u' as observable in W.

We propose to make successive backward traversals from vertices in W. During the backward traversal from $u' \in W$, we label by u' each encountered vertex. At the end of the traversal, the following is true:

- nodes with label u' have u' as observable in W;
- nodes that do not have label u' do not have u' as observable in W.

This labeling will be persistent though iterations. When propagating backwards another vertex $u'' \in W$, we update the ancestors of u'', but preserve the labeling of the other nodes. At the end of the propagation of u'', the following properties are true:

- nodes with label u'' have u'' as observable in W;
- nodes that have a label different from u'' can reach W and do not have u'' as observable in W.

Finding an edge (u, v) such that u has label u'' and v has a label different from u'' proves that u is W-weakly deciding. We can add all such nodes u to W and start another traversal.

We make the following three remarks about the approach mentioned above, before giving a full informal definition of our algorithm.

Remark 7.1. Each node is labeled by zero or one vertex, not by a set of vertices like in Danicic's algorithm (cf. Section 6.5.2). This is the fulfillment of the idea that, to detect a W-critical edge (u, v), we do not need to compute the whole observable sets of u and v; exhibiting one node in $\mathsf{obs}_G(u, W)$ but not in $\mathsf{obs}_G(v, W)$ is enough.

Remark 7.2. The order in which we run through the different actions is really important:

- first, we make the whole backward traversal from a vertex $u' \in W$;
- at the end of the traversal, we detect edges (u, v) such that u is labeled with u' and v has another label;
- once we have detected all such edges, we add their sources to W and update their labels.

The three actions cannot be interleaved, or at least not in a trivial way. Indeed, the property we need from the traversal from u' is that nodes with a label not equal to u' do not have u' as observable vertex in W. This property does not hold while the traversal is not completed. If we try to detect W-critical edges before the traversal is finished, we could identify wrong W-critical edges. For example, consider the graph shown in Figure 7.1. Initially, $W = \{u, v\}$, u is labeled with u, v is labeled with v and every other vertex is unlabeled (see Figure 7.1a). We

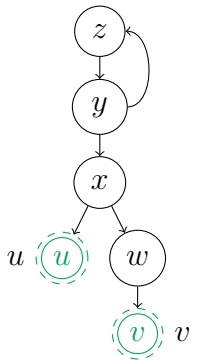
select one node in W that has not been propagated yet. Both u and v are possible choices. We select u and propagate it backwards (see Figure 7.1b). Vertices x, y and z are all labeled with u. Then we detect nodes labeled with u having a child with a different label. We do not find any, thus we start another backward propagation from v. Figure 7.1c illustrates one intermediate state of the graph during the propagation. Nodes w, x and y are labeled with v, but z has not been traversed yet and is still labeled with u. If we apply our criterion in this state, we can detect the edge (y, z) since y has label v and z has a label different from v. This would prove that y is W-weakly deciding. But this is obviously wrong, since every W-path from y contains x. If we let the propagation finish before detecting W-weakly deciding nodes (see Figure 7.1d), we correctly ignore the edge (y, z), since z is labeled with v as well. Instead, we identify the edge (x, u) with x labeled with v and u labeled with u. x is added to W and its label is updated to x (see Figure 7.1e). The next step would be to propagate x backwards, but this is not illustrated here.

Note that we cannot interleave the detection of W-critical edges and the update of W either, since this could again reveal wrong edges. For example, consider Figure 7.1e. If we detect edges such that the source is labeled with v and the target has a label distinct from v in this configuration, where we have already added x to W and updated its label, we identify (y, x) with y labeled with v and x having a label different from v. Thus, to avoid this kind of wrong detection, we update W once we have ended the detection phase.

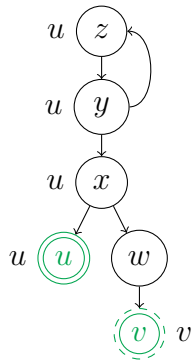
Since the three actions cannot be interleaved, this basically means that we traverse the transitive predecessors of u' three times: once to propagate the label, once to detect W-critical edges and once to update the labels of the sources of the identified edges. This is slightly more complicated than that as it is explained in Section 7.2, but there is certainly room for improvement here.

Remark 7.3. A potential drawback of having a persistent labeling is that it needs to be kept up-to-date. After the propagation of node $u' \in W$, a node is labeled with u' if and only if it has u' as observable vertex in W. But once we have propagated another node $u'' \in W$, what can be said about nodes labeled with u'? Are they still correctly labeled by an observable vertex? Otherwise, do we need to post-process the graph at the end of the propagation to make sure that all the nodes with a label propagated before have a correct label? The answer to both questions is no.

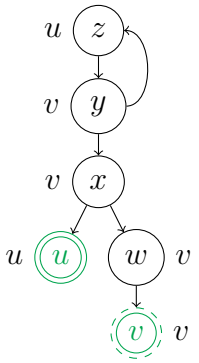
Indeed, let us recall how we detect W-weakly deciding nodes after the propagation of u''. These are nodes u labeled with u'' having a child v with a different label. The label of the child v is required to witness the fact that it can reach W. If the label is still valid (i.e. still an observable vertex), this shows that the child v can reach the node with which it is labeled, thus can reach W. If it is



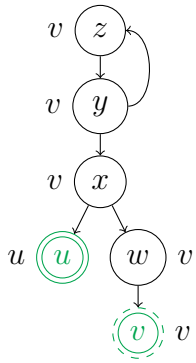
(a) Before any propagation



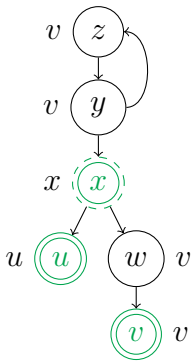
(b) After the propagation of u



(c) During the propagation of v



(d) At the end of the propagation of v



(e) After the propagation of v

vertex in W not processed yet

vertex in W already processed

Figure 7.1 – Execution of the algorithm on an example graph. Initially, $W = \{u, v\}$.

outdated, this means that at least one new node was added to W that hides the node with which the child v is labeled. In this case, the child v can reach at least one of these new nodes and therefore can reach W. Thus, whether the label of the child is outdated or not, it proves that the child can reach W, which is enough to prove that the parent labeled with u'' is W-weakly deciding. Our algorithm thus allows nodes to have outdated labels, and runs correctly despite the presence of such outdated labels.

This is illustrated by Figure 7.1e where y and z both have label v while their only observable in W is x. Nevertheless, at the end of the algorithm, the labeling is correct, in the sense that every vertex labeled is labeled with one of its observable vertices in W (when it exists). Since at the end of the algorithm, we have $W = V' \cup \mathsf{WD}_G(V')$, each node in the graph has at most one observable in W (this follows straightforwardly from Property 6.4). This means that at the end of the algorithm each node is labeled with its observable in W (when it exists). Our algorithm can thus be viewed as returning two results: $V' \cup \mathsf{WD}_G(V')$ that is the main result, and the labeling of each node with its observable in $V' \cup \mathsf{WD}_G(V')$ (when it exists).

7.2 Informal Description of the New Algorithm

The inputs of our algorithm are a graph G and a subset of vertices V' of G. It uses internally three variables:

- a set W, equal to V' initially, that grows during the execution of the algorithm, and is equal to $V' \cup \mathsf{WD}_G(V')$ at the end;
- a partial mapping *obs* associating to each node u in G at most one label $obs[u]$ which is a vertex in W reachable from u and is the unique observable vertex from u in W at the end; initially, *obs* is defined on W and associates each node in W to itself;
- a worklist L of nodes of W not processed yet, initially equal to V'.

As mentioned before, the algorithm is iterative. Each iteration is structured as follows:

- If the worklist L is empty, the algorithm ends.
- Otherwise, L is not empty. A node u is removed from L. Then:
    1. A backward traversal of G from u is performed, so that each vertex that transitively precedes vertex u in G and that is not hidden by vertices in W is labeled with u; during the propagation, the vertices whose labels

have changed (i.e. which had a label before the propagation and are labeled with u after the propagation) and with at least two children are accumulated in a set of candidate W-weakly deciding nodes C;

2. At the end of the propagation, the set of accumulated nodes C is filtered so that only nodes with a child having a label different from u are kept. Such nodes are W-weakly deciding.
3. When all the nodes have been identified, each of them is given itself as a label in *obs*, and is added to W and L.

At the end of the algorithm, W and *obs* are returned and satisfy:

- $W = V' \cup \mathsf{WD}_G(V')$;
- for every node u in G,
  - if u can reach $V' \cup \mathsf{WD}_G(V')$, then its unique observable vertex in $V' \cup \mathsf{WD}_G(V')$ is $obs[u]$;
  - otherwise, u has no label in *obs*, which we denote $u \notin obs$.

The presented algorithm is just a synthetic version of what was discussed in Section 7.1. The only point that was omitted in Section 7.1 is the set C of vertices accumulated during the backward propagation. Actually, the idea is that we can exclude during the propagation some nodes that we are sure will not be identified in the following step of the iteration as W-weakly deciding nodes. This allows to avoid doing the second backward traversal that was discussed in Remark 7.2. We could imagine complex filtering during the propagation, but here we decided to use one of the simplest ones. We remove from the nodes accumulated in C:

- nodes that have only one child, since obviously they cannot be W-weakly deciding;
- nodes that were previously unlabeled. Indeed, if a node is unlabeled at the beginning of the propagation, this means that none of its children was traversed during the past iterations. Thus the children are unlabeled too, except if they are in V' initially and they have not been propagated yet. Thus this filtering is not completely trivial, since with it the algorithm does not behave exactly the same as the algorithm without the filtering. Actually, nodes that are not detected as W-weakly deciding whereas they have a child in V' not propagated yet will be identified in a later iteration, when the child in question will be propagated.

  For example, consider Figure 7.2, where a variant of the graph of Figure 7.1 is shown. x has no label initially (see Figure 7.2a), thus it is not added to

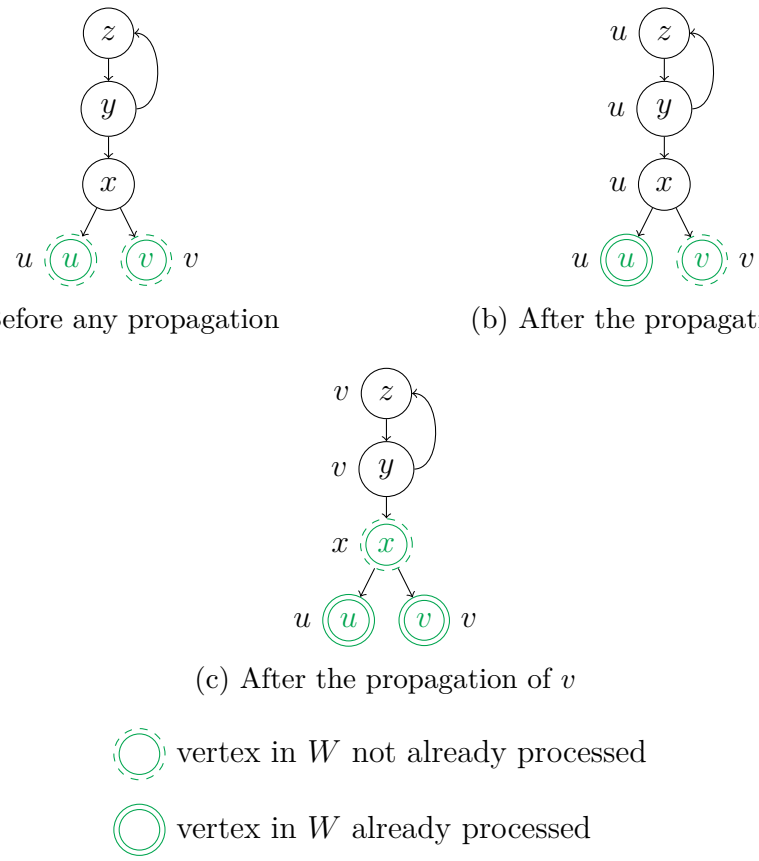


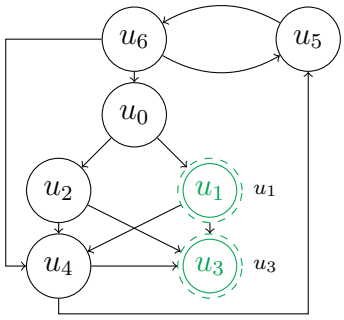
Figure 7.2 – Execution of the algorithm on a variant of the graph shown in Figure 7.1. Initially, $W = \{u, v\}$.

the set of accumulated vertices during the propagation of u and thus is not detected as a W-weakly deciding vertex at the end of the propagation (see Figure 7.2b). However, at the end of the propagation, x is labeled u and it has child v with label v different from u, thus it would be added to W if the filtering was not applied during the propagation. x is actually identified as a W-weakly deciding vertex in the next iteration, where v is propagated (see Figure 7.2c).

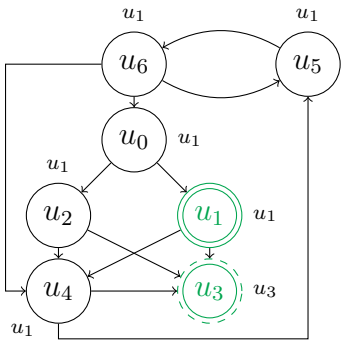
Since the filtering done during the propagation step is really simple, it is unclear whether it really improves the performance of the algorithm. This is studied in Section 9.3.7.

Now that we have informally presented the full algorithm, let us apply it on the running example of Figure 6.3. The execution is illustrated in Figure 7.3.

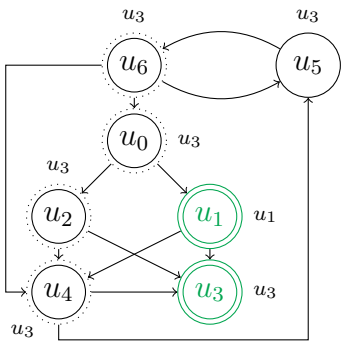
0. Initially (see Figure 7.3a), $W = V' = \{u_1, u_3\}$ and all nodes in W are



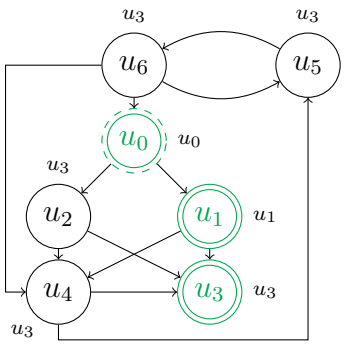
(a) Initially



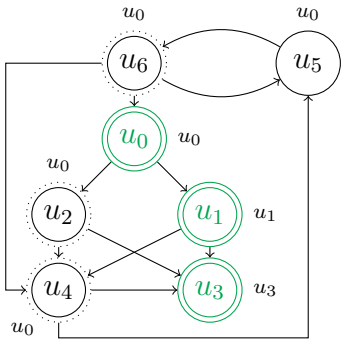
(b) 1st iteration: propagation of u_1



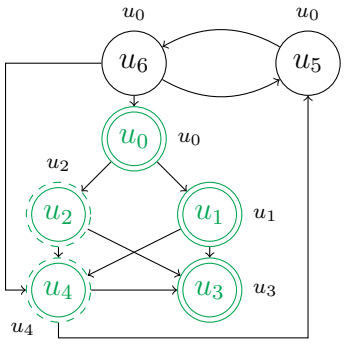
(c) 2nd iteration: propagation of u_3



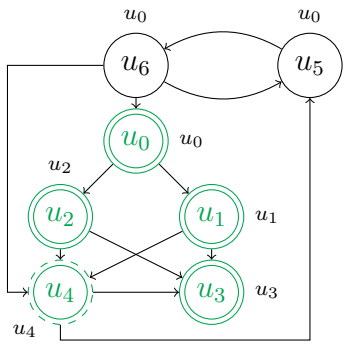
(d) 2nd iteration: detection & update



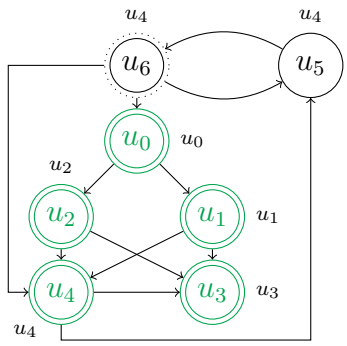
(e) 3rd iteration: propagation of u_0



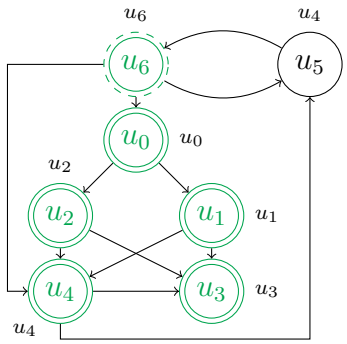
(f) 3rd iteration: detection & update



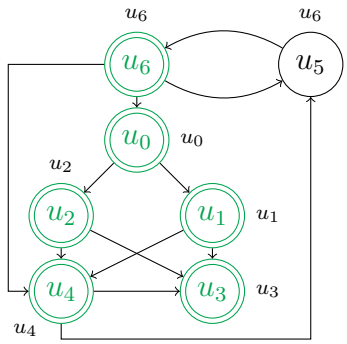
(g) 4th iteration: propagation of u_2



(h) 5th iteration: propagation of u_4



(i) 5th iteration: detection & update



(j) 6th iteration: propagation of u_6

vertex accumulated during the propagation

vertex in W not already processed (i.e. in L)

vertex in W already processed (i.e. in $W \setminus L$)

Figure 7.3 – The optimized algorithm applied on G_0 (cf. Figure 6.3) with $V_0' = \{u_1, u_3\}$

unprocessed, i.e. $L = W = \{u_1, u_3\}$. Each node in W is labeled with itself ($obs[u_1] = u_1$, $obs[u_3] = u_3$).

1. 1st iteration:

   - u_1 is removed from L and is propagated backwards (see Figure 7.3b). Since no vertex outside of W had a label before the propagation, no vertex is accumulated during the propagation, so the iteration ends.

   At the end of the iteration, $W = \{u_1, u_3\}$ and $L = \{u_3\}$.

2. 2nd iteration:

   - u_3 is removed from L and is propagated backwards (see Figure 7.3c). Vertices u_0, u_2, u_4 and u_6 are accumulated during the propagation, since their labels changed and they have at least two children. The label of u_5 changed too, but it has only one child.
   - Among u_0, u_2, u_4 and u_6, only u_0 has a child with a label different from u_3: u_1, that has label u_1. u_0 is thus added to W and L, and its label is updated to itself (see Figure 7.3d).

   At the end of the iteration, $W = \{u_0, u_1, u_3\}$ and $L = \{u_0\}$.

3. 3rd iteration:

   - u_0 is removed from L and is propagated backwards (see Figure 7.3e). Vertices u_2, u_4 and u_6 are accumulated during the propagation, since their labels changed and they have at least two children.
   - Among u_2, u_4 and u_6, vertices u_2 and u_4 have a common child with a label different from u_0: u_3, that has label u_3. Vertices u_2 and u_4 are thus added to W and L, and the label of each of them is updated to itself (see Figure 7.3g).

   At the end of the iteration, $W = \{u_0, u_1, u_2, u_3, u_4\}$ and $L = \{u_2, u_4\}$.

4. 4th iteration:

   - u_2 is removed from L and is propagated backwards (see Figure 7.3g), but its only parent, u_0, is already in W. Thus the propagation ends without having accumulated any node, and the iteration ends.

   At the end of the iteration, $W = \{u_0, u_1, u_2, u_3, u_4\}$ and $L = \{u_4\}$.

5. 5th iteration:

- u_4 is removed from L and is propagated backwards (see Figure 7.3h). u_6 is accumulated during the propagation.
- u_6 has child u_0 with label u_0 different from u_4. u_6 is thus added to W and L, and its label is updated to itself (see Figure 7.3i).

At the end of the iteration, $W = \{u_0, u_1, u_2, u_3, u_4, u_6\}$ and $L = \{u_6\}$.

6. 6th iteration:

   - u_6 is removed from L and is propagated backwards (see Figure 7.3j). No vertex is accumulated during the propagation, thus the iteration ends.

   At the end of the iteration, $W = \{u_0, u_1, u_2, u_3, u_4, u_6\}$ and $L = \emptyset$.

7. 7th iteration: L is empty, thus the algorithm ends.

$W = \{u_0, u_1, u_2, u_3, u_4, u_6\}$ and the labeling *obs* as shown in Figure 7.3j are returned. Since, as mentioned in Section 6.2, every node in G_0 is reachable from V_0', the result of our algorithm is directly equal to the weak control-closure of V_0'. The post-processing mentioned in 7.1 that removes from the final value of W the nodes not reachable from V' does not remove any node here. We can verify that the final value of W is equal to the weak control-closure of V_0' computed using the definition in Section 6.3 and using Danicic's algorithm in Section 6.5.2.

7.3 Formal Definition of the New Algorithm

Our algorithm is split into three functions that are a nearly straightforward implementation of the informal description given in Section 7.2:

- `propagate` takes a vertex and propagates backwards a label over its transitive predecessors. It accumulates a set of vertices whose labels change during the propagation and that have at least two children.
- `confirm` is used to check if a given node has a child with a label different from a given label.
- `main` implements the optimized algorithm using calls to `propagate` and `confirm`. More precisely, as long as there exist unprocessed nodes in the manipulated set W, it selects such a node, marks it as processed, calls `propagate` on it that returns candidate W-weakly deciding nodes, calls `confirm` on them to keep only true W-weakly deciding nodes, adds them to W and updates their labels.

The implementations of these three functions assume that they have access to the following operations on graphs:

- a function `choose` that selects a node in a set of vertices;
- a function `pred` such that `pred` (G, u) returns the set of direct predecessors of vertex u in graph G;
- a function `succ` such that `succ` (G, u) returns the set of children of vertex u in graph G;
- a function `out_degree` such that `out_degree` (G, u) returns the out-degree of vertex u in graph G, i.e. the number of children of u in G.

In the rest of this section, we detail each of the three functions.

7.3.1 Function `propagate`

Function `propagate` takes five arguments, in this order:

- a graph G,
- a subset W of nodes of G,
- a labeling of nodes *obs*, i.e. a partial mapping $obs : V \to V$,
- a vertex u,
- a vertex v.

It traverses G backwards from u (stopping at nodes in W) and updates *obs* so that all transitive predecessors of u not hidden by vertices in W have label v at the end of the function. It returns a set of potential W-weakly deciding vertices accumulated during the propagation.

One can observe that the starting vertex u and the propagated vertex v can be distinct, while in the informal description of the algorithm given in Section 7.2, and this will be confirmed in the formal definition of `main` below, they are systematically the same vertex. We considered such simple generalization since it is natural and does not make the proofs more difficult.

The formal definition of `propagate` is given as Algorithm 7.1. `propagate` manipulates three sets of vertices:

- L, a worklist of nodes to be processed;

- C, the set of nodes whose label changes during the propagation and that have at least two children;
- P, that temporarily stores the set of direct predecessors of the vertex being processed.

Initially, the single node to be processed is u and the set of accumulated nodes C is empty (lines 2–3). The major part of `propagate` is a loop that iterates until no more nodes need to be processed (lines 4–26).

Each iteration can be decomposed as follows:

- a node w is selected in L and removed from L (lines 5–6);
- P is assigned the set of direct predecessors of w in G (line 7);
- for each node w_0 in P (lines 8–25):
  - if w_0 is not in W (line 11),
    - if w_0 has a label (line 12),
      - if this label is different from v (line 13), this means that w_0 has not been traversed by the propagation yet; its label is updated and it is added to the worklist (lines 14–15); moreover, if it has at least two children, it is added to C (lines 16–17);
      - if this label is equal to v, this means that w_0 has already been traversed by the propagation, we stop the propagation in this direction and thus do nothing (line 13);
    - if w_0 is not labeled (line 12), this means that w_0 has not been traversed by the propagation yet; its label is updated and it is added to the worklist (lines 21–22);
  - if w_0 is in W, we stop the propagation in this direction, and thus do nothing (line 11).

When all nodes have been processed, `propagate` ends and returns C the set of nodes whose label changed during the propagation and that have at least two children. Note that `propagate` has side effects, since it modifies the labeling *obs*.

7.3.2 Function `confirm`

Function `confirm` takes four arguments, in this order:

- a graph G,
- a labeling of graph vertices *obs*,

Input: $G = (V, E)$ a directed graph
$W \subseteq V$ a set of nodes ignored by the propagation
obs : Map(V, V) associating at most one label to each vertex
$u \in V$ the vertex where the propagation starts
$v \in V$ the label to propagate
Output: obs modified in place
$C \subseteq V$ containing potential W-weakly deciding nodes

```
1  begin
2      L ← {u}                                          // initialization
3      C ← ∅
4      while L ≠ ∅ do                                   // main loop
5          w ← choose (L)
6          L ← L \ {w}
7          P ← pred (G, w)
8          while P ≠ ∅ do                // browses the predecessors of w
9              w0 ← choose (P)
10             P ← P \ {w0}
11             if w0 ∉ W then
12                 if w0 ∈ obs then
13                     if obs[w0] ≠ v then
14                         obs[w0] ← v
15                         L ← L ∪ {w0}
16                         if out_degree (G, w0) > 1 then
17                             C ← C ∪ {w0}
18                         end
19                     end
20                 else
21                     obs[w0] ← v
22                     L ← L ∪ {w0}
23                 end
24             end
25         end
26     end
27     return C
28 end
```

Algorithm 7.1: Function `propagate` (G, W, obs, u, v)

- a vertex u,
- a vertex u'.

It returns a Boolean. Its result is *true* if and only if at least one child v of u in G has a label in *obs* different from u'.

The formal definition of `confirm` is given as Algorithm 7.2. It is pretty self-explanatory. Boolean b is set to *false* initially (line 2). We traverse the children of u (lines 3–9). If one of them has a label distinct from u' (line 7), we set b to *true*. When all children have been explored, b is returned (line 10).

Input: $G = (V, E)$ a directed graph
$obs : \mathrm{Map}(V, V)$ associating at most one label to each vertex
$u \in V$ the vertex whose children are inspected
$u' \in V$ the label tested against those of the children
Output: $b : bool$, *true* if at least one child of u has a label distinct from u', *false* otherwise

```
1  begin
2      b ← false
3      S ← succ (G, u)
4      while S ≠ ∅ do
5          v ← choose (S)
6          S ← S \ {v}
7          if v ∈ obs and obs[v] ≠ u' then
8              b ← true
9      end
10     return b
11 end
```

Algorithm 7.2: Function `confirm` (G, obs, u, u')

7.3.3 Function `main`

Function `main` takes two arguments, in this order:

- a graph G,
- a subset of vertices V'.

It returns $V' \cup \mathsf{WD}_G(V')$ and a labeling associating to each node its observable vertex in this set if it exists.

The formal definition of `main` is given as Algorithm 7.3. It uses a concise notation to denote batch updates of partial mappings. Given a mapping $obs : \mathrm{Map}(V, V)$, a subset V' of V and a function $f : V' \to V'$, the notation $obs_{|V'} \leftarrow f$ denotes the mapping obs' such that:

$$\forall u \in V, obs'[u] = \begin{cases} f(u) \text{ if } u \in V' \\ obs[u] \text{ otherwise} \end{cases}$$

Given a subset V', we use $id_{V'}$ to denote the identity function restricted to V', i.e.:

$$\begin{aligned} id_{V'} : V' &\to V' \\ u &\mapsto u \end{aligned}$$

By combining these two notations, given a mapping $obs : \mathrm{Map}(V, V)$ and a subset V' of V, we can write $obs_{|V'} \leftarrow id_{V'}$ which is a very concise way to denote obs with the label of each node in V' updated to itself. This is used both in the initialization (line 3) and at each iteration (line 15) of `main`.

`main` manipulates four sets of vertices:

- W, the set V' augmented with V'-deciding vertices;
- L, a worklist of nodes of W not processed yet;
- C, a set of candidate W-weakly deciding vertices;
- Δ, a set of new W-weakly deciding vertices.

Initially, both W and L are equal to V' and each node in V' is given itself as a label in obs (lines 2–4).

The major part of `main` is a loop that iterates until no more nodes need to be processed (lines 5–19).

Each iteration can be decomposed as follows:

- a node u is selected in L and removed from L (lines 6–7);
- label u is propagated backwards from u by calling `propagate` (G, W, obs, u, u) (line 8); the set of candidate W-weakly deciding vertices returned by the call is assigned to C;
- for each node v in C (lines 10–15):
  - if v has a child with a label distinct from u, which is established by calling `confirm`(G, obs, v, u), it is W-weakly deciding, and is thus added to Δ;

```
Input: G = (V, E) a directed graph
       V' ⊆ V the input subset
Output: W ⊆ V the main result
        obs : Map(V, V) the final labeling
begin
    W ← V'                                          // initialization
    obs|V' ← id_V'
    L ← V'
    while L ≠ ∅ do                                  // main loop
        u ← choose(L)
        L ← L \ {u}
        C ← propagate (G, W, obs, u, u)             // propagation
        Δ ← ∅
        while C ≠ ∅ do                              // filtering
            v ← choose(C)
            C ← C \ {v}
            if confirm (G, obs, v, u) = true then
                Δ ← Δ ∪ {v}
        end
        W ← W ∪ Δ                                   // update
        obs|Δ ← id_Δ
        L ← L ∪ Δ
    end
    return (W, obs)
end
```

Algorithm 7.3: Function `main` (G, V')

- L and W are updated. They are added the nodes in Δ. The label of each node in Δ is updated to itself (lines 16–18).

When all nodes have been processed, `main` ends and returns $W = V' \cup \mathsf{WD}_G(V')$ and *obs* the labeling of each vertex by its observable in W if it exists.

Notice that, as stated in Remark 7.2, the main steps (propagation, detection and update) are performed one after the other (respectively line 8, lines 9–15 and lines 16–18).

In this chapter, we presented the main ideas and the exact definition of our optimized algorithm. But we did not provide any proof that it is correct, nor that it is faster than Danicic's. This is addressed by Chapters 8 and 9 respectively.

Chapter 8

Proof of Correctness of the New Algorithm

To have the same level of confidence in the optimized algorithm described in Chapter 6 as in Danicic's algorithm formalized in Coq (cf. Chapter 6), we decided to mechanically prove the optimized version too. Making a proof in Coq was considered, but since our algorithm is more complex than Danicic's, the invariants on which it relies are more subtle. To avoid doing everything manually, we opted for a tool with greater automation: the Why3 proof platform. The automatic provers that Why3 can use as backends automatically discharge the goals that are simple, which allows to focus on the more complex ones that still require manual proofs. Another advantage of Why3 is that it is more suitable than Coq for straightforwardly implementing imperative algorithms, and the description of the algorithm we gave in Section 7.3 is imperative. This last point is illustrated by the excerpts of Why3 code given in Section 8.5 that can be compared to their paper counterparts given in Section 7.3.

To establish the correctness of the algorithm, we prove separately the correctness of the three functions `propagate`, `confirm` and `main`. This consists in annotating them with contracts specifying their preconditions and postconditions, and proving that they respect these contracts. For that, we annotate their bodies with invariants, variants and assertions where needed.

This chapter is organized as follows. Section 8.1 gives contracts to the graph operations listed in Section 7.3. These contracts are needed to prove the correctness of the functions of our algorithm. In Section 8.2, we prove the correctness of function `propagate`. Next, in Section 8.3, we prove the correctness of function `confirm`. Then, in Section 8.4, we use the contracts of `propagate` and `confirm` to prove the correctness of `main`. Section 8.5 presents the Why3 formalization and gives some observations about it.

As discussed in Section 8.5, the optimized algorithm is formalized and proved

correct in Why3. For the sake of completeness, though, this chapter presents the paper-and-pencil version of the proofs automatically or manually done in Why3. Some of these proofs being rather technical, the reader can prefer to refer to the mechanically checked formalization available in [Léc18].

8.1 Contracts of Graph Operations

In Section 7.3, we enumerated four operations on graph that we assumed available: `choose`, `pred`, `succ` and `out_degree`. Since these functions are called by `propagate`, `confirm` and `main`, we need to give them contracts before proving the latter.

The contract of `choose`, given as Algorithm 8.1 is straightforward. If the set of vertices V given as an argument is non-empty, `choose` (V) returns a vertex in V.

Input: V a set of vertices
Output: u a vertex
Requires: $V \neq \emptyset$
Ensures: $u \in V$

Algorithm 8.1: Contract of function `choose` (V)

The contracts of `pred` and `succ` are similar and are based on a mathematical definition of predecessors and successors respectively.

Definition 8.1

Given a directed graph $G = (V, E)$ and a node $u \in V$, we define the set of predecessors of u in G, denoted $\mathsf{pred}_G(u)$, and the set of successors of u in G, denoted $\mathsf{succ}_G(u)$, respectively as:

- $\mathsf{pred}_G(u) = \{v \in V \mid (v, u) \in E\}$.
- $\mathsf{succ}_G(u) = \{v \in V \mid (u, v) \in E\}$.

Based on Definition 8.1, the contracts of `pred` and `succ` just assert that `pred` and `succ` are correct implementations of pred_G and succ_G. Their contracts are given as Algorithms 8.2 and 8.3 respectively.

The contract of `out_degree` is given as Algorithm 8.4. Given a graph G and a node u in G, `out_degree` (G, u) returns the number of successors of u in G, i.e. $|\mathsf{succ}_G(u)|$.

Input: $G = (V, E)$ a directed graph
$u \in V$ a vertex
Output: $X \subseteq V$
Ensures: $X = \mathsf{pred}_G(u)$

Algorithm 8.2: Contract of function `pred` (G, u)

Input: $G = (V, E)$ a directed graph
$u \in V$ a vertex
Output: $X \subseteq V$
Ensures: $X = \mathsf{succ}_G(u)$

Algorithm 8.3: Contract of function `succ` (G, u)

Now that we have a contract for those four functions, we can prove the correctness of the functions of the algorithm.

8.2 Proof of Correctness of `propagate`

First, we need to attach a contract to `propagate`. This contract is given as Algorithm 8.5.

The contract contains two preconditions that must be satisfied when calling `propagate` (G, W, obs, u, v):

($\mathbf{P_1}$) Initially, u has label v and it is the only vertex to have label v. This is required to ensure that the propagation does not stop too early. Indeed, recall that `propagate` detects that a node has already been traversed by looking at its label and testing whether it is equal to v (cf. Algorithm 7.1). If initially some nodes already have label v, `propagate` will stop and will not explore their predecessors.

($\mathbf{P_2}$) Initially, vertex u is in W. Since W is not modified by `propagate`, this is true during the execution of the function and at its end. This precondition

Input: $G = (V, E)$ a directed graph
$u \in V$ a vertex
Output: $n \in \mathbb{N}$
Ensures: $n = |\mathsf{succ}_G(u)|$

Algorithm 8.4: Contract of function `out_degree` (G, u)

is needed to formulate the postconditions this way, since by Definition 6.8, for any $z \in V$, $z \xrightarrow{W-path} u$ implies that u is in W.

The contract contains three postconditions that are established by a call to `propagate` (G, W, obs, u, v), if $(\mathbf{P_1})$ and $(\mathbf{P_2})$ held before the call:

$(\mathbf{Q_1})$ All nodes in V that have u as observable vertex in W are labeled with v at the end of the function.

$(\mathbf{Q_2})$ All other nodes have the same label as before the call.

$(\mathbf{Q_3})$ At the end of the call, the set C contains exactly all the nodes that are distinct from u, have u as observable vertex in W, had a label *before* the call and have at least two children. This postcondition expresses formally the filtering operated during the propagation that is discussed in Section 7.2.

Input: $G = (V, E)$ a directed graph
$W \subseteq V$ a set of nodes ignored by the propagation
$obs : \mathrm{Map}(V, V)$ associating at most one label to each vertex
$u \in V$ the vertex where the propagation starts
$v \in V$ the label to propagate
Output: obs modified in place (called obs' in the postconditions)
$C \subseteq V$ containing all the possible conflicts of obs'
Requires: $(\mathbf{P_1})$ $\forall z \in V, obs[z] = v \iff z = u$
Requires: $(\mathbf{P_2})$ $u \in W$
Ensures: $(\mathbf{Q_1})$ $\forall z \in V, z \xrightarrow{W-path} u \implies obs'[z] = v$
Ensures: $(\mathbf{Q_2})$ $\forall z \in V, \neg(z \xrightarrow{W-path} u) \implies obs'[z] = obs[z]$
Ensures: $(\mathbf{Q_3})$ $\forall z \in V, z \in C \iff z \neq u \wedge z \xrightarrow{W-path} u \wedge z \in obs$
$\wedge |\mathsf{succ}_G(z)| > 1$

Algorithm 8.5: Contract of function `propagate` (G, W, obs, u, v)

Now that we have given a contract to `propagate`, we need to prove that it is respected by the function. Classically, we annotate the body of `propagate` with invariants, variants and assertions, as described in Section 2.2.1 in the context of Why3. The correctness of `propagate` was unexpectedly hard to establish. Actually, proving that all the nodes traversed are relabeled is rather easy, but proving that we traversed all of the transitive predecessors of the starting vertex turned out to be difficult.

Algorithm 8.6 presents the body of function `propagate` with annotations. The explicit definitions of the annotations used in Algorithm 8.6 are given in Figure 8.1.

Compared to the body of `propagate` without annotations (cf. Algorithm 7.1), we add invariants and variants to the loops at line 4 and at line 8, and two assertions at the end of the program before the `return` instruction at line 27. Note that some invariants are shared by both loops: $(\mathbf{I_1})$, $(\mathbf{I_2})$, $(\mathbf{I_4})$, $(\mathbf{I_5})$ and $(\mathbf{I_6})$. We also added two labels, namely `INIT` at the beginning of the program and `LOOP` before the inner loop at line 8, that are used in the invariants (e.g. $(\mathbf{I_5})$) to refer to the values hold by variables of the program at these program points. Formally, we use the expression $\mathbf{at}(e, L)$ to designate the value of expression e at label L. For example, $\mathbf{at}(obs, \texttt{INIT})$ denotes the value of variable *obs* at the beginning of the program, which is in fact the value passed as an argument when `propagate` is called.

In the rest of this section, we present the annotations one by one and prove that they are correct. Finally, we prove that they imply postconditions $(\mathbf{Q_1})$, $(\mathbf{Q_2})$ and $(\mathbf{Q_3})$.

Let us start with the invariants.

Proof of invariant $(\mathbf{I_1})$: $\forall z \in V, obs[z] = v \implies z \xrightarrow{W-path} u$.[1]
Invariant $(\mathbf{I_1})$ states that each vertex that has label v has u as observable vertex in W.

- It is true initially, since by $(\mathbf{P_1})$ only u has v as label and u is observable from itself in W.
- Assume it holds at the beginning of an iteration of the outer loop at line 4.
  - Then it holds at the entry of the inner loop at line 8.
  - Assume that it holds at the beginning of an iteration of the inner loop. To prove that it is true at the end of the iteration of the inner loop, it is sufficient to prove that the updates to *obs* at lines 14 and 21 preserve the property, i.e. that w_0 has indeed u as observable in W. Both updates are performed in the then-branch of the condition at line 11, thus $w_0 \notin W$. Moreover, since w_0 was extracted from P (line 9), it is thus a parent of w (by invariant $(\mathbf{I_7})$). w was itself extracted from L (line 5), thus is labeled with u (by invariant $(\mathbf{I_4})$), and thus there exists a W-path $w \xrightarrow{W-path} u$ by $(\mathbf{I_1})$ at the beginning of the iteration of the inner loop. By prepending the edge (w_0, w) to this W-path, we exhibit a W-path between w_0 and u. This proves that $(\mathbf{I_1})$ holds at the end of the iteration of the inner loop.

[1] The mechanized version of this proof is available in [Léc18].

```
1  begin
       INIT:
2      L ← {u}                                          // initialization
3      C ← ∅
4      while L ≠ ∅ do                                   // main loop
           // invariant: I1 ∧ I2 ∧ I3 ∧ I4 ∧ I5 ∧ I6
           // variant: (|{z ∈ V | obs[z] ≠ v}|, |L|)
5          w ← choose (L)
6          L ← L \ {w}
7          P ← pred (G, w)
           LOOP:
8          while P ≠ ∅ do            // browses the predecessors of w
               // invariant:  I1 ∧ I2 ∧ I'3 ∧ I''3 ∧ I4 ∧ I5 ∧ I6 ∧ I7 ∧ I8 ∧ I9
               // variant: |P|
9              w0 ← choose (P)
10             P ← P \ {w0}
11             if w0 ∉ W then
12                 if w0 ∈ obs then
13                     if obs[w0] ≠ v then
14                         obs[w0] ← v
15                         L ← L ∪ {w0}
16                         if out_degree (G, w0) > 1 then
17                             C ← C ∪ {w0}
18                         end
19                     end
20                 else
21                     obs[w0] ← v
22                     L ← L ∪ {w0}
23                 end
24             end
25         end
26     end
       // assert: A1 ∧ A2
27     return C
28 end
```

Algorithm 8.6: Annotated body of `propagate` (G, W, obs, u, v) (cf. Algorithm 7.1)

$(\mathbf{I_1})$ $\forall z \in V, obs[z] = v \implies z \xrightarrow{W-path} u$

$(\mathbf{I_2})$ $obs[u] = v$

$(\mathbf{I_3})$ $\forall z \in V, obs[z] = v \wedge z \notin L$
$\implies \forall z' \in V, z' \in \mathsf{pred}_G(z) \wedge z' \notin W \implies obs[z'] = v$

$(\mathbf{I_4})$ $\forall z \in L, obs[z] = v$

$(\mathbf{I_5})$ $\forall z \in V, obs[z] \neq v \implies obs[z] = \mathbf{at}(obs[z], \mathtt{INIT})$

$(\mathbf{I_6})$ $\forall z \in V, z \in C \iff z \neq u \wedge obs[z] = v \wedge z \in \mathbf{at}(obs, \mathtt{INIT})$
$\wedge\, |\mathsf{succ}_G(z)| > 1$

$(\mathbf{I'_3})$ $\forall z \in V, obs[z] = v \wedge z \notin L \wedge z \neq w$
$\implies \forall z' \in V', z' \in \mathsf{pred}_G(z) \wedge z' \notin W \implies obs[z'] = v$

$(\mathbf{I''_3})$ $\forall z \in \mathtt{pred}_G(w), z \notin P \wedge z \notin W \implies obs[z] = v$

$(\mathbf{I_7})$ $P \subseteq \mathsf{pred}_G(w)$

$(\mathbf{I_8})$ $\forall z \in V, \mathbf{at}(obs[z], \mathtt{LOOP}) = v \implies obs[z] = v$

$(\mathbf{I_9})$ $(\forall z \in V, obs[z] = v \implies \mathbf{at}(obs[z], \mathtt{LOOP}) = v)$
$\implies L = \mathbf{at}(L, \mathtt{LOOP})$

$(\mathbf{A_1})$ $\forall z \in V, obs[z] = v \implies z \xrightarrow{W-path} u$

$(\mathbf{A_2})$ $\forall z \in V, z \xrightarrow{W-path} u \implies obs[z] = v$

Figure 8.1 – Definitions of the invariants and assertions annotating the body of `propagate` (G, W, obs, u, v) (cf. Algorithm 8.6)

Thus $(\mathbf{I_1})$ is preserved by the inner loop, and therefore holds at the end of the iteration of the outer loop. □

Proof of invariant $(\mathbf{I_2})$: $obs[u] = v$.[2]
Invariant $(\mathbf{I_2})$ states that u has label v throughout the iterations.

- It is true initially thanks to precondition $(\mathbf{P_1})$.
- Assume it is true at the beginning of an iteration of the outer loop.
  - Then it is true at the entry of the inner loop.
  - Assume it is true at the beginning of an iteration of the inner loop. It is sufficient to show that during the updates at lines 14 and 21, $w_0 \neq u$. This last point is true, since by condition at line 11, $w_0 \notin W$, while by precondition $(\mathbf{P_2})$, $u \in W$. Thus u has label v at the end of the iteration of the inner loop.

  Thus $(\mathbf{I_2})$ is preserved by the inner loop, and therefore holds at the end of the iteration of the outer loop. □

Proof of invariant $(\mathbf{I_3})$: $\forall z \in V, obs[z] = v \wedge z \notin L$
$\implies \forall z' \in V, z' \in \mathsf{pred}_G(z) \wedge z' \notin W \implies obs[z'] = v$.[3]
Invariant $(\mathbf{I_3})$ states that if a vertex has label v and is not in worklist L, then it has already been processed, which means that all its parents not in W have also label v.

- It is vacuously true initially, since u is the only vertex with label v by $(\mathbf{P_1})$, and $L = \{u\}$.
- Assume it is true at the beginning of an iteration of the outer loop. We show that the invariants:

  $(\mathbf{I'_3})$ $\forall z \in V, obs[z] = v \wedge z \notin L \wedge z \neq w$
  $\implies \forall z' \in V', z' \in \mathsf{pred}_G(z) \wedge z' \notin W \implies obs[z'] = v$

  and

  $(\mathbf{I''_3})$ $\forall z \in \mathtt{pred}_G(w), z \notin P \wedge z \notin W \implies obs[z] = v$

  are valid invariants of the inner loop. $(\mathbf{I'_3})$ is nearly identical to $(\mathbf{I_3})$, but excludes w from the considered nodes. $(\mathbf{I''_3})$ describes the labels of the parents of w. All the parents that have already been processed and that are not in W have label v.

[2]The mechanized version of this proof is available in [Léc18].
[3]The mechanized version of this proof is available in [Léc18].

- In the entry of the inner loop, $(\mathbf{I'_3})$ holds by $(\mathbf{I_3})$ at the beginning of the iteration of the outer loop, since the exclusion of w in $(\mathbf{I'_3})$ compensates for the removal of w from L at line 6. $(\mathbf{I''_3})$ is vacuously true, since $P = \texttt{pred}_G(w)$ by the postcondition of `pred`.
- Assume that $(\mathbf{I'_3})$ and $(\mathbf{I''_3})$ are true at the beginning of an iteration of the inner loop. At the end of the iteration, $(\mathbf{I'_3})$ holds for $z = w$, since it is explicitly excluded; it holds for $z = w_0$, since w_0 was added to L; and it holds for the other vertices, since w_0 cannot be the parent of a node distinct from w, that has label v and is not in L by $(\mathbf{I'_3})$ at the beginning of the iteration. Thus $(\mathbf{I'_3})$ holds at the end of the iteration.

  $(\mathbf{I''_3})$ straightforwardly holds at the end of the iteration, since if w_0 is not in W, its label is set to v at line 14 or at line 21.

Thus $(\mathbf{I'_3})$ and $(\mathbf{I''_3})$ are preserved by the inner loop. In particular, they hold at the end of the inner loop. By combining them, we prove exactly $(\mathbf{I_3})$, by applying $(\mathbf{I'_3})$ if $z \neq w$ or $(\mathbf{I''_3})$ if $z = w$. Thus $(\mathbf{I_3})$ holds at the end of the execution of the outer loop. □

Proof of invariant $(\mathbf{I_4}) : \forall z \in L, obs[z] = v$.[4]
Invariant $(\mathbf{I_4})$ states that each node in the worklist L has label v.

- This is true initially, since $L = \{u\}$ and u has label v by $(\mathbf{P_1})$.
- Assume that this is true at the beginning of an iteration of the outer loop.
  - Then it is true at the entry of the inner loop. Indeed, if every node in L has label v, *a fortiori* each node in $L \setminus \{w\}$ has label v.
  - Assume that $(\mathbf{I_4})$ holds at the beginning of an iteration of the inner loop. When w_0 is added to L at line 15 or 22, its label has just been set to v (line 14 or 21). Thus $(\mathbf{I_4})$ holds at the end of the iteration of the inner loop.

  Thus $(\mathbf{I_4})$ is preserved by the inner loop, and therefore holds at the end of the iteration of the outer loop. □

Proof of invariant $(\mathbf{I_5}) : \forall z \in V, obs[z] \neq v \implies obs[z] = \mathbf{at}(obs[z], \texttt{INIT})$.[5]
Invariant $(\mathbf{I_5})$ states that the only change in the labeling *obs* that `propagate` is allowed to perform is to set labels to v. If a node has no label or a label distinct from v, then its labeling has not been modified during the propagation.

[4] The mechanized version of this proof is available in [Léc18].
[5] The mechanized version of this proof is available in [Léc18].

- This is true initially, since $obs = \mathbf{at}(obs, \texttt{INIT})$.
- Assume that $(\mathbf{I_5})$ holds at the beginning of an iteration of the outer loop.
  - Then it holds at the entry of the inner loop.
  - Assume that $(\mathbf{I_5})$ holds at the beginning of an iteration of the inner loop. The only change applied to *obs* is to potentially set the label of w_0 to v. Thus $(\mathbf{I_5})$ is weaker at the end of the iteration than at the beginning. Since it holds at the beginning, it holds at the end.

  Thus $(\mathbf{I_5})$ is preserved by the inner loop, and therefore holds at the end of the iteration of the outer loop. □

Proof of invariant $(\mathbf{I_6})$: $\forall z \in V, z \in C \iff z \neq u \wedge obs[z] = v$
$\wedge z \in \mathbf{at}(obs, \texttt{INIT})$
$\wedge |\mathsf{succ}_G(z)| > 1$.[6]

Invariant $(\mathbf{I_6})$ states that the nodes in C are exactly the nodes that are distinct from u, have v as observable, had a label before the call to `propagate` and have at least two children.

- This is true initially, since $C = \emptyset$ and u is the only node that has v as label by $(\mathbf{P_1})$.
- Assume that $(\mathbf{I_6})$ is true at the beginning of an iteration of the outer loop.
  - Then it holds at the entry of the inner loop.
  - Assume that it is true at the beginning of an iteration of the inner loop. We prove it is also true at the end of the iteration, by proving successively the implication and its converse.
    - Let z be a vertex in C at the end of the iteration of the inner loop.
      - Assume that z was already in C at the beginning of the iteration of the inner loop. By $(\mathbf{I_6})$ at the beginning of the iteration, $z \neq u$, z had label v at the beginning of the iteration, z had a label before the call to `propagate` and z has at least two children. The possible changes to the labeling *obs* during the iteration of the inner loop (lines 14 and 21) both update some label to v. Thus z still has label v at the end of the iteration.

[6] The mechanized version of this proof is available in [Léc18].

· Assume that z was not in C at the beginning of the iteration. This implies that it was added on line 17. Therefore, $z = w_0$, $w_0 \notin W$, w_0 had a label at the beginning of the iteration and this label was not equal to v. We can make the following observations. First, since w_0 had not label v at the beginning of the iteration of the inner loop, $w_0 \neq u$ by $(\mathbf{I_4})$. Second, the label of w_0 was updated to v at line 14. Third, since w_0 had a label distinct from v at the beginning of the iteration, w_0 had the same label in *obs* before the call to `propagate` (by $(\mathbf{I_5})$). Fourth, by the postcondition of `out_degree`, $|\mathsf{succ}_G(w_0)| > 1$.

Thus, at the end of the iteration:

$$\forall z \in V, z \in C \implies z \neq u \wedge obs[z] = v \wedge z \in at(obs, \texttt{INIT}) \\ \wedge |\mathsf{succ}_G(z)| > 1$$

∗ Let z be a vertex in V such that $z \neq u$, z is labeled with v at the end of the iteration of the inner loop, z had a label before the call to `propagate` and has at least two children. If z had already label v at the beginning of the iteration, by $(\mathbf{I_6})$ at the beginning of the iteration, $z \in C$. If z did not have label v before the iteration, this means that $z = w_0$ and its label was updated on line 14 or on line 21. If its label was updated on line 14, since z has at least two children, z was added to C on line 17 by the postcondition of `out_degree`. If its label was updated on line 21, this means that z did not have a label at the beginning of the iteration. But this implies that z did not have a label before the call to `propagate` either (by $(\mathbf{I_5})$), which contradicts the hypotheses about z. Thus, at the end of the iteration:

$$\forall z \in V, z \neq u \wedge obs[z] = v \wedge z \in at(obs, \texttt{INIT}) \\ \wedge |\mathsf{succ}_G(z)| > 1 \implies z \in C$$

By combining both results, we get that $(\mathbf{I_6})$ holds at the end of the iteration of the inner loop.

Thus $(\mathbf{I_6})$ is preserved by the inner loop, and therefore holds at the end of the iteration of the outer loop. □

Proof of invariant $(\mathbf{I_7})$: $P \subseteq \mathsf{pred}_G(w)$.[7]
Invariant $(\mathbf{I_7})$ states that P contains only parents of w in G.

- When entering the inner loop, this is true since $P = \mathsf{pred}_G(w)$ by the postcondition of `pred`.

[7] The mechanized version of this proof is available in [Léc18].

- After an iteration, P is smaller than at the beginning of the iteration. Thus, by the transitivity of inclusion, if $(\mathbf{I_7})$ holds at the beginning of the iteration, it also holds at its end. □

Proof of invariant $(\mathbf{I_8})$: $\forall z \in V, \mathbf{at}(obs[z], \texttt{LOOP}) = v \implies obs[z] = v$.[8]
Invariant $(\mathbf{I_8})$ states that nodes that already have label v when entering the inner loop still have label v when leaving the loop.

- When entering the inner loop, this is true since $obs = \mathbf{at}(obs, \texttt{LOOP})$.
- During an iteration of the inner loop, the only label that may change (line 14 or 21) is set to v. Thus, if $(\mathbf{I_8})$ holds at the beginning of the iteration, it also holds at its end. □

Proof of invariant $(\mathbf{I_9})$: $(\forall z \in V, obs[z] = v \implies \mathbf{at}(obs[z], \texttt{LOOP}) = v)$
$\implies L = \mathbf{at}(L, \texttt{LOOP})$.[9]
Invariant $(\mathbf{I_9})$ states that if, after some iterations of the inner loop, all the nodes that have label v already had label v when entering the inner loop, then L contains the same elements as when entering the inner loop.

- When entering the inner loop, this is true since $L = \mathbf{at}(L, \texttt{LOOP})$.
- Assume that $(\mathbf{I_9})$ holds at the beginning of an iteration of the inner loop. If, at the end of the iteration, there exists a node $z \in V$ that has label v whereas this was not the case at the beginning of the iteration, then $(\mathbf{I_9})$ trivially holds, since the left part of the implication is false. On the contrary, if all the nodes that have label v already had label v at the beginning of the iteration, this means that $obs[w_0]$ has not changed and thus, since the updates to L are coupled to the changes to $obs[w_0]$, L contains the same elements as at the beginning of the iteration. By $(\mathbf{I_9})$ at the beginning of the iteration, L contains the same elements as when entering the inner loop. □

Thus all the invariants are valid. Let us prove that both loops terminate, by proving the validity of the corresponding variants.

Proof of the variant of the outer loop: $(|\{z \in V \mid obs[z] \neq v\}|, |L|)$.[10]
This is a lexicographic variant. The first component is the cardinality of the set

[8] The mechanized version of this proof is available in [Léc18].
[9] The mechanized version of this proof is available in [Léc18].
[10] The mechanized version of this proof is available in [Léc18].

containing all vertices in G with a label distinct from v. The second part is the size of the worklist L. The idea of this variant is that during the processing of node w extracted from worklist L, the number of nodes with label v grows. But it is possible that this number does not strictly increase, in the case where all the parents of w are in W or have already label v. In that case, though, no node is added to L, and since w has just been removed from it, L is strictly smaller than at the beginning of the iteration of the outer loop.

Let us prove that this variant is correct formally. To prove that it is compatible with the lexicographical order on $\mathbb{N}^2$, we prove that it is not smaller than $(0, 0)$ and that it strictly decreases during an iteration.

- Both components are cardinalities, thus are non-negative.
- To prove that the variant decreased after one iteration, we use invariants $(\mathbf{I_8})$ and $(\mathbf{I_9})$ that were tailored to that purpose. Indeed $(\mathbf{I_8})$ states that the number of vertices with label v does not decrease during an iteration, which means that the number of vertices with label v does not increase during an iteration. In the case where the number of vertices with label v remains equal, then $(\mathbf{I_9})$ comes to the rescue, by proving that L has the same elements as when entering the inner loop. Since w was removed from L on line 6, L is strictly smaller at the end of the iteration of the outer loop than at its beginning. This shows that the variant at the end of the iteration is strictly smaller (with respect to the lexicographical order on $\mathbb{N}^2$) than at its beginning. □

Proof of the variant of the inner loop: $|P|$.[11]
This second variant is much simpler. It is the size of set P. Since it is the cardinality of a set, it is non-negative. In each iteration of the inner loop, one element is removed from P, thus the variant strictly decreases during an iteration of the inner loop. □

We can now prove the two assertions and the postconditions.

$(\mathbf{A_1})$ states that after the call to `propagate` (G, W, obs, u, v) every node labeled with v has u as observable vertex in W. This is proved straightforwardly by invariant $(\mathbf{I_1})$.

$(\mathbf{A_2})$ states that after the call to `propagate` (G, W, obs, u, v) every node that has u as observable vertex in W has label v. This is proved using invariants $(\mathbf{I_2})$ and $(\mathbf{I_3})$. Indeed, since L is empty when leaving the outer loop, $(\mathbf{I_3})$ is equivalent to the following:

$$\forall z \in V, obs[z] = v \implies \forall z' \in V, z' \in \mathsf{pred}_G(z) \wedge z' \notin W \implies obs[z'] = v$$

[11] The mechanized version of this proof is available in [Léc18].

By $(\mathbf{I_2})$, u has label v, and by $(\mathbf{I_3})$, if a node has label v, then all its parents not in W have also label v. By a simple induction, we can prove that every node $z \in V$ such that $z \xrightarrow{W-path} u$ has label v.

Postcondition $(\mathbf{Q_1})$ is exactly assertion $(\mathbf{A_2})$, it is therefore true.

To prove $(\mathbf{Q_2})$, we use $(\mathbf{I_5})$ that states that each node whose label is not v has the same label as before the call. By combining $(\mathbf{A_1})$ and $(\mathbf{A_2})$, we prove that the nodes that are labeled with v after the call are exactly the nodes that have u in their observable set in W. This implies that the nodes that are not labeled v are the nodes that do not have u as observable vertex in W, which proves $(\mathbf{Q_2})$.

From $(\mathbf{I_6})$ and by observing again that the nodes with label v are the nodes connected to u with W-path, we can deduce $(\mathbf{Q_3})$. This terminates the proof that `propagate` fulfills its contract.

8.3 Proof of Correctness of `confirm`

Compared to the proof of `propagate`, the proof of `confirm` is quite simple. A version of `confirm` annotated with a contract, invariants and variants is given as Algorithm 8.7.

`confirm` does not have any precondition, and its single postcondition states that it returns *true* if and only if one of the successors v of u has a label v' distinct from u'.

The first invariant states that the set of tested nodes S is a subset of the children of u. This is true initially since $S = \mathtt{succ}(G, u)$ and remains true after an iteration since S decreases.

The second invariant is identical to the postcondition except that it considers only nodes that have been already explored, i.e. nodes in $\mathsf{succ}_G(u) \setminus S$. This is true initially, since $b = \mathit{false}$ and $\mathsf{succ}_G(u) \setminus S = \emptyset$ by the postcondition of `succ`. This is preserved after one iteration, since the invariant describes exactly what is tested in the while body.

The variant that proves the termination of the loop is the cardinality of set S. Indeed, it is non-negative and strictly decreases during one iteration since an element is removed from S.

8.4 Proof of Correctness of `main`

Now that we have given contracts to `propagate` and `confirm` and proved that they are respected by the functions, we can do the same for function `main`. Algorithm 8.8 presents function `main` annotated with a contract, invariants, variants

Input: $G = (V, E)$ a directed graph
$obs : \mathrm{Map}(V, V)$ associating at most one label to each vertex
$u \in V$ the vertex whose children are inspected
$u' \in V$ the label tested against those of the children
Output: $b : bool$, $true$ if at least one child of u has a label distinct from u', $false$ otherwise
Ensures: $b = true \iff \exists v, v' \in V, v \in \mathsf{succ}_G(u) \wedge obs[v] = v' \wedge v' \neq u'$

```
1  begin
2      b ← false
3      S ← succ (G, u)
4      while S ≠ ∅ do
           // invariant: S ⊆ succ_G(u)
           // invariant: b = true ⟺ ∃v, v' ∈ V, v ∈ (succ_G(u) \ S)
                                     ∧obs[v] = v' ∧ v' ≠ u'
           // variant: |S|
5          v ← choose (S)
6          S ← S \ {v}
7          if v ∈ obs and obs[v] ≠ u' then
8              b ← true
9      end
10     return b
11 end
```

Algorithm 8.7: Function `confirm` (G, obs, u, u') with annotations

and assertions. The explicit definitions of the invariants and the assertions are given in Figure 8.2.

Let us first focus on the contract. No precondition is attached to `main`. Unsurprisingly, there are two postconditions:

- the first one states that set W returned by `main` is equal to $V' \cup \mathsf{WD}_G(W)$;
- the second one states that the labeling returned by `main` is correct, in the sense that each node is labeled by its observable when it exists.

To prove the two postconditions, we attached invariants and variants to the loops on line 5 and on line 10, and inserted four assertions before the `return` instruction on line 20. We describe all of them hereafter. Like for `propagate`, we inserted a label `LOOP` before the inner loop on line 10, that allows to refer to the values hold by variables of the program at that program point.

Proof of invariant $(\mathbf{I_1}) : L \subseteq W$**.**[12]
Invariant $(\mathbf{I_1})$ states that each vertex in the worklist L is also in W.

- This is true initially since $W = L = V'$.
- Assume that $L \subseteq W$ at the beginning of an iteration. At the end of the iteration, the new values of L and W are:

  $$L' = (L \setminus \{u\}) \cup \Delta \text{ and } W' = W \cup \Delta$$

  Since $L \subseteq W$, we have $L \setminus \{u\} \subseteq W$, and thus $L \setminus \{u\}) \cup \Delta \subseteq W \cup \Delta$, i.e. $L' \subseteq W'$. This proves that $(\mathbf{I_1})$ holds at the end of the iteration. □

Proof of invariant $(\mathbf{I_2}) : \forall z \in W, obs[z] = z$**.**[13]
Invariant $(\mathbf{I_2})$ states that each node in W has itself as a label.

- It is true initially since $W = V'$ (line 2) and each node in V' has itself as a label (line 3).
- Assume that $(\mathbf{I_2})$ holds at the beginning of an iteration. During the updates of W and *obs* (lines 16 and 17), the nodes that are added to W have their label updated to themselves. And by postcondition $(\mathbf{Q_2})$ of `propagate`, nodes in W have the same label at the end of the iteration as at its beginning, thus have themselves as a label by $(\mathbf{I_2})$ at the beginning of the iteration. Thus $(\mathbf{I_2})$ holds at the end of the iteration. □

[12] The mechanized version of this proof is available in [Léc18].
[13] The mechanized version of this proof is available in [Léc18].

Input: $G = (V, E)$ a directed graph
$V' \subseteq V$ the input subset
Output: $W \subseteq V$ the main result
$obs : \mathrm{Map}(V, V)$ the final labeling
Ensures: $W = V' \cup \mathsf{WD}_G(V')$
Ensures: $\forall u, v \in V, obs[u] = v \iff v \in \mathsf{obs}_G(u, W)$

```
1  begin
2      W ← V'                                          // initialization
3      obs|V' ← id_V'
4      L ← V'
5      while L ≠ ∅ do                                  // main loop
           // invariant: I1 ∧ I2 ∧ I3 ∧ I4 ∧ I5 ∧ I6 ∧ I7
           // variant: (|V \ W|, |L|)
6          u ← choose(L)
7          L ← L \ {u}
8          C ← propagate (G, W, obs, u, u)             // propagation
9          Δ ← ∅
           LOOP:
10         while C ≠ ∅ do                              // filtering
               // invariant: I'6 ∧ I'7 ∧ I8
               // variant: |C|
11             v ← choose(C)
12             C ← C \ {v}
13             if confirm (G, obs, v, u) = true then
14                 Δ ← Δ ∪ {v}
15         end
16         W ← W ∪ Δ                                   // update
17         obs|Δ ← id_Δ
18         L ← L ∪ Δ
19     end
       // assert: A1 ∧ A2 ∧ A3 ∧ A4
20     return (W, obs)
21 end
```

Algorithm 8.8: Function `main` (G, V') with annotations

$(\mathbf{I_1})$ $L \subseteq W$

$(\mathbf{I_2})$ $\forall z \in W, obs[z] = z$

$(\mathbf{I_3})$ $\forall z, z' \in V, obs[z] = z' \implies z' \in W$

$(\mathbf{I_4})$ $\forall z, z' \in V, obs[z] = z' \wedge z' \in L \implies z = z'$

$(\mathbf{I_5})$ $\forall z, z' \in V, obs[z] = z' \implies z \xrightarrow{path} z'$

$(\mathbf{I_6})$ $V' \subseteq W \subseteq V' \cup \mathsf{WD}_G(V')$

$(\mathbf{I_7})$ $\forall y, z, z' \in V, y \xrightarrow{W-disjoint} z \wedge obs[z] = z' \wedge z' \notin L \implies obs[y] = z'$

$(\mathbf{I'_6})$ $\Delta \subseteq \mathsf{WD}_G(W)$

$(\mathbf{I'_7})$ $\forall y, z, z' \in V,$
$y \notin C \wedge y \neq u \wedge y \xrightarrow{W-path} u \wedge z \in \mathsf{succ}_G(y) \wedge obs[z] = z' \wedge z' \notin L \cup \{u\}$
$\implies y \in \Delta$

$(\mathbf{I_8})$ $C \subseteq \mathbf{at}(C, \mathtt{LOOP})$

$(\mathbf{A_1})$ $\forall y, z \in V, y \xrightarrow{W-path} z \implies obs[y] = z$

$(\mathbf{A_2})$ $\mathsf{WD}_G(W) = \emptyset$

$(\mathbf{A_3})$ $V' \subseteq W \subseteq V' \cup \mathsf{WD}_G(V')$

$(\mathbf{A_4})$ $W = V' \cup \mathsf{WD}_G(V')$

Figure 8.2 – Definitions of the invariants and assertions annotating the body of function `main` (G, V') (cf. Algorithm 8.8)

Proof of invariant $(\mathbf{I_3})$: $\forall z, z' \in V, obs[z] = z' \implies z' \in W$.[14]
Invariant $(\mathbf{I_3})$ states that all labels are in W.

- This is true initially since $W = V'$ (line 2) and all labels are in V' (line 3).
- Assume that $(\mathbf{I_3})$ is true at the beginning of an iteration. At the end of the iteration, the new value of W is $W' = W \cup \Delta$ and the new labeling obs' satisfies:

$$\forall z \in V, obs'[z] = \begin{cases} z & \text{if } z \in \Delta \\ u & \text{if } z \notin \Delta \wedge z \xrightarrow{W-path} u \\ obs[z] & \text{if } \neg(z \xrightarrow{W-path} u) \end{cases}$$

  Let z be a vertex with label z'. There are three possible cases:

  - if $z' \in \Delta$, then $z' \in W'$ since $\Delta \subseteq W'$;
  - if $z' = u$, we can observe that u is extracted from L on line 7, thus it is in W by $(\mathbf{I_1})$; therefore $z' \in W$, whence $z' \in W'$;
  - if z' is the label of z at the beginning of the iteration, then $z' \in W$ by $(\mathbf{I_3})$ at the beginning of the iteration; thus, $z' \in W'$.

  Therefore, $(\mathbf{I_3})$ holds at the end of the iteration. □

Proof of invariant $(\mathbf{I_4})$: $\forall z, z' \in V, obs[z] = z' \wedge z' \in L \implies z = z'$.[15]
Invariant $(\mathbf{I_4})$ states that labels in L have not been propagated yet. Given a node z in L, z is the only node whose label is z.

- This is true initially since $L = V'$ (line 3), all the nodes that are labeled are in V' and each of these nodes has itself as a label.
- Assume that $(\mathbf{I_4})$ holds at the beginning of an iteration, i.e.

$$\forall z, z' \in V, obs[z] = z' \wedge z' \in L \implies z = z'$$

  Let L' and obs' be the values of L and obs respectively at the end of the iteration. As stated above,

$$L' = (L \setminus \{u\}) \cup \Delta$$

  Let z and z' be two vertices in V. Let us assume that $obs'[z] = z'$ and $z' \in L'$. By invariant $(\mathbf{I'_6})$ of the inner loop (proved below), $\Delta \subseteq \mathsf{WD}_G(W)$, and thus, by Property 6.2, $\Delta \cap W = \emptyset$. Since, by invariant $(\mathbf{I_1})$, $L \subseteq W$, this means that L' is the disjoint union of $L \setminus \{u\}$ and Δ. Thus, either $z' \in L \setminus \{u\}$ or $z' \in \Delta$.

[14] The mechanized version of this proof is available in [Léc18].
[15] The mechanized version of this proof is available in [Léc18].

- Assume that $z' \in L \backslash \{u\}$. By postconditions $(\mathbf{Q_1})$ and $(\mathbf{Q_2})$ of `propagate`, the nodes that have a label in $L \setminus \{u\}$ at the end of the iteration are the same as at the beginning of the iteration. Thus, $obs[z] = obs'[z] = z'$, and by $(\mathbf{I_4})$ at the beginning of the iteration, $z = z'$.
- Assume that $z' \in \Delta$. By $(\mathbf{I_3})$ at the beginning of the iteration, all the labels at the beginning of the iteration are in W and W is disjoint from Δ by $(\mathbf{I'_6})$. Thus, the label of z has just been updated to z' in this iteration. $z' \neq u$, since $u \in W$, thus the last time the label of z was modified was not the call to `propagate`. This means the label of z was updated on line 17. Thus, z is in Δ and $z = z'$.

This proves that $(\mathbf{I_4})$ holds at the end of the iteration. □

Proof of invariant $(\mathbf{I_5})$: $\forall z, z' \in V, obs[z] = z' \implies z \xrightarrow{path} z'$.[16]
Invariant $(\mathbf{I_5})$ states that if label z' is associated to a node z then there exists a path between z and z'.

- Initially, there exists a trivial path from each node in V' to itself, thus $(\mathbf{I_5})$ holds.
- Assume that $(\mathbf{I_5})$ holds at the beginning of an iteration. Let z be a vertex in V. At the end of the propagation, by postconditions $(\mathbf{Q_1})$ and $(\mathbf{Q_2})$ of `propagate`, the new labeling obs' satisfies:

$$\forall z \in V, obs'[z] = \begin{cases} z & \text{if } z \in \Delta \\ u & \text{if } z \notin \Delta \wedge z \xrightarrow{W-path} u \\ obs[z] & \text{if } \neg(z \xrightarrow{W-path} u) \end{cases}$$

  We explore the three different cases.

  - If $z \in \Delta$ and $obs'[z] = z$, there is a path from z to z.
  - If $z \notin \Delta \wedge z \xrightarrow{W-path} u$ and $obs'[z] = u$, there exists a W-path from z to u and thus in particular a path from z to u.
  - If $\neg(z \xrightarrow{W-path} u)$ and $obs'[z] = obs[z]$, we have a path between z and z' by $(\mathbf{I_5})$ at the beginning of the iteration. Indeed, during the iteration, we change sets W and L and labeling obs, but not the graph itself. Thus, if there exists a path between two nodes at a given iteration, it remains a valid path in all following iterations.

[16] The mechanized version of this proof is available in [Léc18].

Thus $(\mathbf{I_5})$ holds at the end of the iteration. □

Proof of invariant $(\mathbf{I_6})$: $V' \subseteq W \subseteq V' \cup \mathsf{WD}_G(V')$.[17]
Invariant $(\mathbf{I_6})$ states that W remains between V' and $V' \cup \mathsf{WD}_G(V')$ during the execution of the algorithm.

- This is true initially, since $W = V'$.
- Assume that $V' \subseteq W \subseteq V' \cup \mathsf{WD}_G(V')$ at the beginning of an iteration and let us prove that $V' \subseteq W' \subseteq V' \cup \mathsf{WD}_G(V')$ at the end of the iteration, where W' is the value of W at the end of the iteration, i.e. $W' = W \cup \Delta$. Since $V' \subseteq W$, we have trivially $V' \subseteq W'$. To conclude, it is thus sufficient to prove that $W' \subseteq V' \cup \mathsf{WD}_G(V')$. Since, by hypothesis, $W \subseteq V' \cup \mathsf{WD}_G(V')$, it is sufficient to prove that $\Delta \subseteq V' \cup \mathsf{WD}_G(V')$.

  To prove it, we use the fact that, since $W \subseteq V' \cup \mathsf{WD}_G(V')$, we have, by Properties 6.3 and 6.4,

  $$\mathsf{WD}_G(W) \subseteq V' \cup \mathsf{WD}_G(V')$$

  It is thus sufficient to prove that $\Delta \subseteq \mathsf{WD}_G(W)$. This is established by the invariant:

  $(\mathbf{I'_6})$ $\Delta \subseteq \mathsf{WD}_G(W)$

  of the inner loop on line 10. Let us prove that this invariant is valid for the inner loop.

  - When entering the inner loop, $\Delta = \emptyset$, thus $(\mathbf{I'_6})$ holds at that point.
  - After an iteration, we must prove that the vertex v that has possibly been added to Δ is W-weakly deciding. We know that v is taken from C. By $(\mathbf{I_8})$, C is a subset of the value of C when entering the inner loop, thus by postcondition $(\mathbf{Q_3})$ of `propagate`:

    $$v \neq u \wedge v \xrightarrow{W-path} u \wedge v \in obs \wedge |\,\mathsf{succ}_G(v)| > 1$$

    Moreover, the call to `confirm` on line 13 returned *true*, which means by the postcondition of `confirm` that v has a child w with label w' distinct from u.

    To prove that v is W-weakly deciding, we exhibit two V'-paths from v that share no vertex except v. The first one is the W-path from v to

[17] The mechanized version of this proof is available in [Léc18].

u given by the postcondition of `propagate`. The second one requires a bit more work. Since w has a label, there exists a path from w to W by invariant $(\mathbf{I_5})$ at the beginning of the iteration of the outer loop. If we consider the prefix of that path until its first intersection with W, we get a W-path from w. By prepending to it the edge (v, w), we get the second W-path from v. Both W-paths do not intersect except in v, since otherwise we would have a W-path $w \xrightarrow{W-path} u$, and thus the label of w after the propagation would have been u and not w', which is contradictory. These two W-paths prove that $v \in \mathsf{WD}_G(W)$. This proves that $(\mathbf{I'_6})$ holds at the end of the iteration of the inner loop.

Using $(\mathbf{I'_6})$, it is straightforward to conclude that

$$V' \subseteq W \cup \Delta \subseteq V' \cup \mathsf{WD}_G(V')$$

i.e. that $(\mathbf{I_6})$ holds at the end of the iteration of the outer loop. □

Proof of invariant $(\mathbf{I_7})$: $\forall y, z, z' \in V, y \xrightarrow{W-disjoint} z \wedge obs[z] = z' \wedge z' \notin L \implies obs[y] = z'$.[18]

Invariant $(\mathbf{I_7})$ is by far the most complicated and the least obvious invariant. It states that if there is a path between two vertices y and z that does not intersect W, i.e. a W-disjoint path (cf. Definition 6.13), and z has a label already processed, then y and z have the same label. It is stronger than one could expect. Indeed, a more natural invariant would be similar to assertion $(\mathbf{A_1})$, i.e.:

$$\forall y, z \in V, y \xrightarrow{W-path} z \wedge z \notin L \implies obs[y] = z$$

This formula is indeed an invariant, since it is implied by $(\mathbf{I_7})$ (by considering $z \in W \setminus L$, the W-disjoint path between y and z is also a W-path, and $z' = z$ by invariant $(\mathbf{I_2})$). But it lacks a precious information about nodes that have already been propagated: if a node has a label already propagated, and its parent is not in W, then its parent has the same label. Transitively, all the transitive predecessors of the node that are not in W have the same label, whence the notion of W-disjoint path in the definition of $(\mathbf{I_7})$.

The fact that a node not in W has the same label as its child if this label has already been processed is one of the key arguments that explain the correctness of the algorithm. Indeed, during the propagation, we select as candidate nodes only those which had already a label before the propagation, and only afterwards we apply `confirm` to detect true W-weakly deciding nodes. This means that we

[18] The mechanized version of this proof is available in [Léc18].

miss the W-weakly deciding vertices that would have been detected by `confirm` if they had not been filtered out during the propagation, as discussed in Section 7.2. But invariant ($\mathbf{I_7}$) guarantees that there are some that will not be missed. Indeed, given a node traversed during the propagation, if the call to `confirm` on it detects a child with a label already processed, then it had the same label at the beginning of the iteration, which means it is added to the set of candidate nodes during the propagation, and thus is detected as a W-weakly deciding node despite the filtering performed during the propagation. It is therefore possible, because of the filtering performed during the propagation, to miss some nodes, but not those with a label already processed at the beginning of the iteration. Informally, invariant ($\mathbf{I_7}$) gives a lower bound of the progress of the algorithm.

Let us prove that ($\mathbf{I_7}$) is a valid invariant.

- Initially, $W = L = V'$, and each node in V' is labeled with itself. Thus any label is in the worklist L, which means that ($\mathbf{I_7}$) is vacuously true.

- Assume that ($\mathbf{I_7}$) holds at the beginning of an iteration. Let W', L' and obs' the values of W, L and obs respectively at the end of the iteration. We recall that $W' = W \cup \Delta$, $L' = (L \setminus \{u\}) \cup \Delta$ and

$$\forall z \in V, obs'[z] = \begin{cases} z & \text{if } z \in \Delta \\ u & \text{if } z \notin \Delta \wedge z \xrightarrow{W-path} u \\ obs[z] & \text{if } \neg(z \xrightarrow{W-path} u) \end{cases}$$

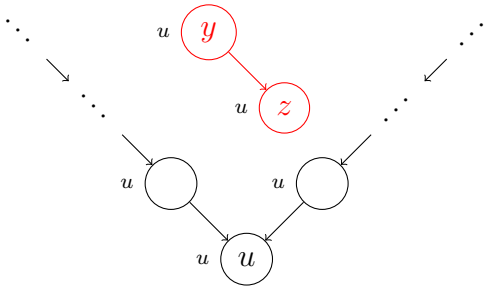
  Let y, z and z' be nodes in V. We assume that $y \xrightarrow{(W\cup\Delta)-disjoint} z$, $obs'[z] = z'$ and $z' \notin L'$. Let us prove that $obs'[y] = z'$.

  First, observe that z is not in Δ. Let us reason by contradiction and assume that $z \in \Delta$. By definition of obs', $obs'[z] = z$, which means that $z' = z$. Thus, z' is in Δ. But we also have in our hypotheses that $z' \notin (L \setminus \{u\}) \cup \Delta$, which is contradictory. Thus z is not in Δ.

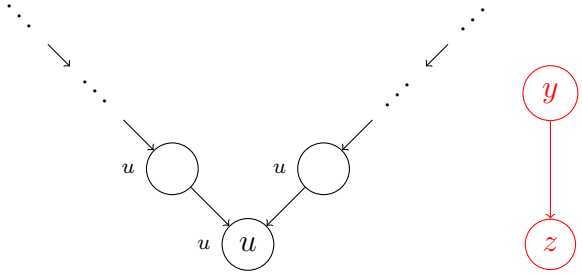
  Likewise, we can prove that y is not in Δ either, otherwise the $(W \cup \Delta)$-disjoint path between y and z would be trivial, i.e. $y = z$. Thus, z would be in Δ, which we have just shown is contradictory.

  We examine three cases depending on whether there exist a W-path between z and u and a W-path between y and u. The three cases are represented in Figure 8.3.

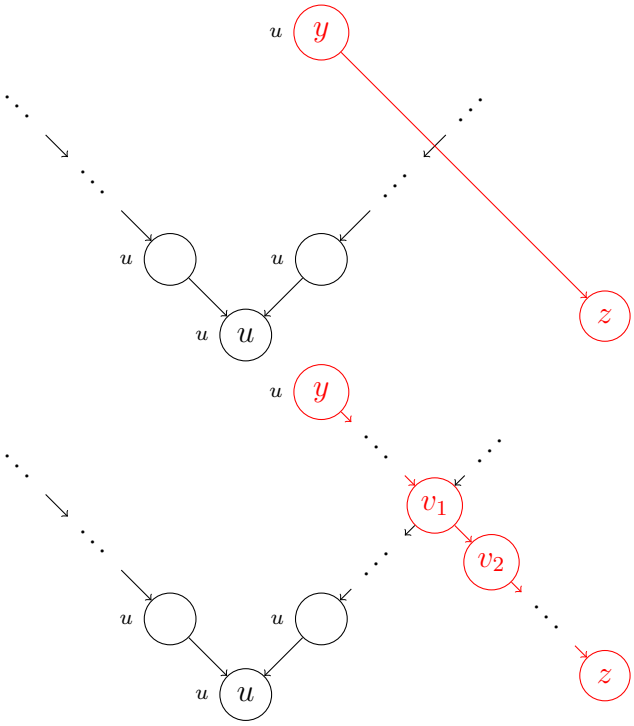
  - Let us assume that $z \xrightarrow{W-path} u$. Since $y \xrightarrow{(W\cup\Delta)-disjoint} z$, we have in particular $y \xrightarrow{W-disjoint} z$. By concatenating that W-disjoint path



(a) Both y and z have just been relabeled with u



(b) Both y and z have the same label as at the beginning of the iteration



(c) y has been relabeled with u, but not z

Figure 8.3 – The three possible configurations in the proof of $(\mathbf{I_7})$

between y and z and the W-path between z and u, we get a W-path between y and u. This is represented in Figure 8.3a. Thus, both y and z have been relabeled with u during the propagation, i.e. $obs'[y] = u$ and $obs'[z] = u$. Thus, we have $obs'[y] = z' = u$.

– Let us assume that there does not exist any W-path from z to u nor any W-path from y to u. This is represented in Figure 8.3b. The labels y and z were not changed by the propagation, i.e. $obs'[y] = obs[y]$ and $obs'[z] = obs[z]$. Moreover, $y \xrightarrow{W-disjoint} z$, since it is a consequence of $y \xrightarrow{(W\cup\Delta)-disjoint} z$. Thus, we can conclude by $(\mathbf{I_7})$ at the beginning of the iteration, provided that we show that z' is not in L. By assumption, z' is not in $(L\setminus\{u\})\cup\Delta$. It is thus sufficient to prove that $z' \neq u$. Let us assume that $z' = u$, i.e. $obs[z] = u$. At the beginning of the iteration, u has not been propagated yet (by $(\mathbf{I_4})$), thus $z = u$, but this contradicts that there does not exist any W-path between z and u. Thus $z' \neq u$, and $obs'[y] = obs[y] = z'$.

– Let us assume that there exists a W-path between y and u, but none between z and u. This is represented by the first part of Figure 8.3c. This means that the label of y was updated to u during the propagation, i.e. $obs'[y] = u$, but z has the same label as before the propagation, i.e. $obs'[z] = obs[z] = z'$. We show that this case is contradictory.

  Consider the $(W \cup \Delta)$-disjoint path between y and z. y is connected to u using a W-path, but z is not. We introduce v_1 as the last vertex on the $(W\cup\Delta)$-disjoint path connecting y and z that is also the origin of a W-path to u, and v_2 as its successor on this $(W \cup \Delta)$-disjoint path. v_1 and v_2 are represented in the second part of Figure 8.3c. We show that $v_1 \in \Delta$, which contradicts the fact that it lies on the $(W \cup \Delta)$-disjoint path from y to z.

  To prove that v_1 is in Δ, we make use of the invariant:

  $(\mathbf{I'_7})$ $\forall y, z, z' \in V,$
  $y \notin C \wedge y \neq u \wedge y \xrightarrow{W-path} u \wedge z \in \mathsf{succ}_G(y) \wedge obs[z] = z' \wedge z' \notin L \cup \{u\}$
  $\implies y \in \Delta$

  of the inner loop. First, we prove that $(\mathbf{I'_7})$ is a valid invariant of the inner loop. Then, we apply it to deduce that v_1 is in Δ.

  $(\mathbf{I'_7})$ states that Δ contains all the nodes not in C and distinct from u that have u as observable vertex in W and have a child with a label not in $L \cup \{u\}$. Note that, here, L denotes the value of L at the entry of the inner loop. This means that u is not in L, since it was removed from it on line 7. The expression $L \cup \{u\}$ is thus the value of L at the

beginning of the iteration of the outer loop, before u was removed from it. Hereafter, we prefer to use L to refer to its value at the beginning of the iteration of the outer loop. We write therefore L instead of $L \cup \{u\}$.

* When entering the inner loop on line 10, to prove that $(\mathbf{I'_7})$ holds, we prove that, after the propagation, each node y such that $y \neq u$, $y \xrightarrow{W-path} u$ and that has a child z with label z' not in L (where L denotes the value of L at the beginning of the outer loop, in particular $u \in L$), is in C. By postcondition $(\mathbf{Q_3})$ of `propagate`, it is enough to prove that $y \neq u$, $y \xrightarrow{W-path} u$, y had a label before the propagation and has at least two children. The first two points are in the hypotheses, so they are trivially true.
  To prove the other two points, let us observe that there is no W-path from z to u. Indeed, assume that there is one. This means that z was traversed during the propagation, and thus its label z' is equal to u. But this is contradictory, since z' is not in L but u is. Thus there is no W-path between z and u. This means in particular that z has the same label as before the propagation.
  The third point is proved thanks to $(\mathbf{I_7})$ at the beginning of the iteration of the outer loop applied to y and its child z. Indeed, from $y \neq u$ and $y \xrightarrow{W-path} u$, we can deduce that y is not in W. Thus y, z is a W-disjoint path between y and z. Moreover, z had label z' not in L before the propagation. By $(\mathbf{I_7})$, the label of y before the propagation was z'. In particular, y had a label before the propagation.
  For the fourth point, consider z and the second vertex on the W-path from y to u. Both are children of y and they cannot be equal, otherwise there would exist a W-path from z to u.
  All this proves that y is in C after the propagation. We can conclude that $(\mathbf{I'_7})$ holds when entering the inner loop.

* Let us assume that $(\mathbf{I'_7})$ is true at the beginning of an iteration of the inner loop. Let y be a vertex such that $y \notin C \setminus \{v\}$, $y \neq u$, $y \xrightarrow{W-path} u$ and there exists a child z of y with label z' not in L (where again L denotes the value of L at the beginning of the outer loop). Let us prove that y is in Δ', where Δ' is the value of Δ at the end of the execution, i.e. $\Delta \cup \{v\}$ if the call to `confirm` on line 13 returned *true* and Δ otherwise. By hypothesis, $y \notin C \setminus \{v\}$, thus $y \notin C$ or $y = v$. If $y \notin C$, we can apply $(\mathbf{I'_7})$ that holds at the beginning of the iteration and deduce that $y \in \Delta$, and thus $y \in \Delta'$. Let us assume that $y = v$, and let us show that the call to `confirm`

on line 13 returned *true*. By the postcondition of `confirm`, it is enough to prove that v has a child with a label different from u. This follows easily from the existence of the child z of $y = v$ with label z' not in L and thus distinct from u, since $u \in L$.
Since the call to `confirm` returned *true*, $\Delta' = \Delta \cup \{v\}$ and thus $v \in \Delta'$. This proves that $(\mathbf{I'_7})$ holds at the end of the iteration of the inner loop.

Thus $(\mathbf{I'_7})$ is preserved by the inner loop. In particular, it holds at the end of the inner loop, where $C = \emptyset$, which gives:

$$\begin{aligned}\forall y, z, z' \in V,\ y \neq u \wedge y \xrightarrow{W-path} u \wedge z \in \mathsf{succ}_G(y)\\ \wedge\ obs[z] = z' \wedge z' \notin L\\ \implies y \in \Delta\end{aligned}$$

Let us apply this result to deduce that v_1 is in Δ. We use it with $y = v_1$, $z = v_2$ and $z' = z'$. It is thus sufficient to prove that:

* $v_1 \neq u$: this is true, since v_1 lies on a $(W \cup \Delta)$-path, and $u \in W$;
* $v_1 \xrightarrow{W-path} u$: this holds by definition of v_1;
* $v_2 \in \mathsf{succ}_G(v_1)$: this is true by definition of v_1 and v_2;
* $obs[v_2] = z'$: by definition of v_2, there is a $(W \cup \Delta)$-disjoint path from v_2 to z, thus in particular a W-disjoint path from v_2 to z. Therefore, by $(\mathbf{I_7})$ at the beginning of the iteration, $obs[v_2] = z'$;
* $z' \notin L$: this holds by hypothesis about z'.

Thus v_1 is in Δ, which contradicts its definition as a vertex on the $(W \cup \Delta)$-disjoint path from y to z. This shows that this third case is contradictory.

Thus $(\mathbf{I_7})$ holds at the end of the iteration of the outer loop.

This concludes the proof that $(\mathbf{I_7})$ is a valid invariant for the outer loop. □

Proof of invariant $(\mathbf{I_8}) : C \subseteq \mathbf{at}(C, \texttt{LOOP})$.[19]
Invariant $(\mathbf{I_8})$ states that set C after some iterations of the inner loop is a subset of set C when entering the inner loop, which is equal to the value returned by `propagate` on line 8.

- This is true when entering the loop, since $C = \mathbf{at}(C, \texttt{LOOP})$.

[19]The mechanized version of this proof is available in [Léc18].

- After an iteration, C is smaller than at the beginning of the iteration. Thus, by the transitivity of inclusion, if $(\mathbf{I_8})$ holds at the beginning of the iteration, it also holds at its end. □

Thus all the invariants are valid. Let us prove that both loops terminate. For that, we prove the validity of the corresponding variants.

Proof of the variant of the outer loop: $(|V \setminus W|, |L|)$.[20]
This is a lexicographic variant that resembles a lot the variant of the outer loop of function `propagate`. The first component is the cardinality of the set of all the vertices in G that are not in W. The second component is the cardinality of the worklist L. Both components are cardinalities, thus are non-negative. Moreover, during an iteration, the vertices in Δ are added to W, and Δ is disjoint from W by invariant $(\mathbf{I'_6})$ of the inner loop, thus W grows. If $\Delta \neq \emptyset$, it grows strictly. If $\Delta = \emptyset$, it remains the same. But in this case, L is smaller at the end of the iteration than at its beginning by one vertex, u. This shows that the variant at the end of the iteration is strictly smaller (with respect to the lexicographical order on $\mathbb{N}^2$) than at its beginning. □

Proof of the variant of the inner loop: $|C|$.[21]
This variant is the size of set C. Since it is the cardinality of a set, it is non-negative. In each iteration of the inner loop, one element is removed from C, thus the variant strictly decreases during an iteration of the inner loop. □

We can now prove the four assertions and deduce from them and the invariants the two postconditions.

$(\mathbf{A_1})$ states that if there exists a W-path between two nodes y and z at the end of function `main`, then y is labeled with z. This is a consequence of invariant $(\mathbf{I_7})$. Let y and z be two nodes verifying $y \xrightarrow{W-path} z$. The W-path between y and z is in particular a W-disjoint path. Moreover, by invariant $(\mathbf{I_2})$, since $z \in W$, we have $obs[z] = z$. We can also note that $z \notin L$, since at the end of `main`, $L = \emptyset$. By $(\mathbf{I_7})$ at the end of the outer loop, $obs[y] = z$.

$(\mathbf{A_2})$ states that there is no W-weakly deciding node. This is actually a direct consequence of $(\mathbf{A_1})$. Indeed, $(\mathbf{A_1})$ implies that each vertex y has at most one observable in W, equal to $obs[y]$ when it exists. A W-weakly deciding vertex would have two observable vertices, thus $\mathsf{WD}_G(W) = \emptyset$.

$(\mathbf{A_3})$ states that, at the end of `main`, V' is a subset of W which is itself a subset of $V' \cup \mathsf{WD}_G(V')$. It is the direct consequence of $(\mathbf{I_5})$ at the end of the outer loop.

[20] The mechanized version of this proof is available in [Léc18].
[21] The mechanized version of this proof is available in [Léc18].

($\mathbf{A_4}$) states that W at the end of `main` is, as expected, equal to $V' \cup \mathsf{WD}_G(V')$. It can be deduced from ($\mathbf{A_2}$) and Property 6.3 applied to ($\mathbf{A_3}$). Indeed, since $V' \subseteq W$, we have $\mathsf{WD}_G(V') \subseteq W \cup \mathsf{WD}_G(W)$, i.e. $\mathsf{WD}_G(V') \subseteq W$ since $\mathsf{WD}_G(W) = \emptyset$ by ($\mathbf{A_2}$). Thus

$$\mathsf{WD}_G(V') \subseteq W \subseteq V' \cup \mathsf{WD}_G(V')$$

Since $V' \subseteq W$,

$$V' \cup \mathsf{WD}_G(V') \subseteq W \subseteq V' \cup \mathsf{WD}_G(V')$$

This gives $W = V' \cup \mathsf{WD}_G(V')$.

Let us now prove the two postconditions.

The first postcondition is exactly assertion ($\mathbf{A_4}$), thus it holds.

The second postcondition is a consequence of assertion ($\mathbf{A_1}$) and invariant ($\mathbf{I_5}$). ($\mathbf{A_1}$) is itself the left-to-right direction of the equivalence. Let us prove the right-to-left direction. Let u, v be two nodes such that $obs[u] = v$, and let us prove that $v \in \mathsf{obs}_G(u, W)$. By ($\mathbf{I_5}$), there is a path from u to v. Let w be the first element in W on this path. Then $u \xrightarrow{W-path} w$. By ($\mathbf{A_1}$), $obs[u] = w$. Thus, $w = v$ and $u \xrightarrow{W-path} v$. This proves the second postcondition.

The two postconditions of `main` prove that, from an initial set V', it constructs correctly the set $V' \cup \mathsf{WD}_G(V')$ and labels each node with its observable vertex in $V' \cup \mathsf{WD}_G(V')$ when it exists. This shows that the optimized algorithm we propose is correct.

8.5 Remarks about the Why3 Formalization

The three functions forming our algorithm, `propagate`, `confirm` and `main`, have been implemented and proved correct in the Why3 proof platform [Léc18]. As mentioned at the beginning of this chapter, their implementations in WhyML, along with their contracts and the annotations used to prove their correctness are very similar to what has been presented in the previous and the current chapters.

An overview of the formalization in Why3 is presented in Section 8.5.1. Then, Section 8.5.2 describes the proof effort. Next, Section 8.5.3 highlights the differences between the Why3 formalization and the formalization presented in the previous chapter and this one. The possibility of extraction of the Why3 formalization into OCaml is discussed in Section 8.5.4.

8.5.1 Overview of the Development

The Why3 formalization has two parts.

```
type graph
type vertex

val eq_vertex (v1 v2 : vertex) : bool
  ensures { result = True <-> v1 = v2 }

function support graph : set vertex

val function succ graph vertex : set vertex
axiom succ_support : forall g u. mem u (support g)
                     -> subset (succ g u) (support g)

val function pred graph vertex : set vertex
axiom pred_succ : forall g u v. mem v (succ g u)
                  <-> mem u (pred g v)
axiom pred_support : forall g u. mem u (support g)
                     -> subset (pred g u) (support g)

val out_degree (g : graph) (u : vertex) : int
  ensures { result = cardinal (succ g u) }
```

Figure 8.4 – The modeling of graphs in the Why3 formalization

- The first part is the set of definitions needed for the algorithm. This consists of the modeling of graphs and a subset of the concepts related to weak control-closure presented in Chapter 6. These are needed to write the algorithm, its specification and to prove the algorithm correct with respect to its specification. This part contains 7 lines of code and 52 lines of specification.
- The second part is the algorithm itself, i.e. the definitions of the three functions `propagate`, `confirm` and `main`. This part consists of 122 lines of code and 124 lines of specification.

Representation of graphs. We formalize finite graphs using a basic representation. We just introduce the few operations that we need. In particular, we introduce Why3 versions of three of the graph functions presented in Section 7.3: `pred`, `succ` and `out_degree` (the fourth function, `choose`, is provided by a Why3 library, see Section 8.5.3).

The Why3 definitions are given in Figure 8.4. We assume the existence of two abstract types `graph` and `vertex` and of some functions manipulating them:

- A function `eq_vertex` is assumed that checks whether two vertices are equal,

as specified by its postcondition.

- We assume the existence of a logical function `support` that models the finite set of nodes in a graph. It is supposed to return from a graph the finite set of vertices in that graph.

- A function `succ` is assumed that models the edge relation of a graph. It is supposed to return the (finite) set of children of a node in a graph. Actually, the keywords **`val function`** introduce both this function `succ` (the Why3 version of `succ` introduced in Section 7.3) and its logical counterpart also named `succ` (the Why3 version of succ_G). Axiom `succ_support` claims that the successors of a node in the graph are also in the graph.

- Similarly to `succ`, the identifier `pred` denotes simultaneously a function (the Why3 version of `pred` introduced in Section 7.3) and its logical counterpart (the Why3 version of pred_G). It is supposed to return the (finite) set of parents of a node in a graph. Its behavior is specified by axiom `pred_succ` with respect to the behavior of `succ`: a node u is a parent of a node v if and only if v is a child of u. Like `succ`, all the parents of a node in the graph are also in the graph as stated by `pred_support`.

- Last, a function `out_degree` (the Why3 version of `out_degree`) is introduced. Its postcondition guarantees that it returns the number of successors of a node in the graph.

Graph definitions. Based on this representation of graphs, several concepts are defined: the notions of path (`path`) and V'-disjoint path (`v_disjoint`) that are both defined as inductive definitions based on `succ`. From that, we can define the notion of V'-path (`v_path`), observable set in V' (`obs_g`) and V'-weakly deciding nodes (`wd`). For instance, the definition of `wd` is given as Figure 8.5. Axiom `wd_spec` specifies its meaning: a node `u` is `v'`-weakly-deciding if and only if there exist two paths `l1` and `l2` from `u` that share only vertex `u` and that end in set `v'` in two distinct points `u1` and `u2`.

Graph properties. We state and prove several properties about the objects that we have just defined. Most of the basic lemmas are proved by the automatic provers, but the most complex ones are proved manually in Coq (see Section 8.5.2). Actually, the results proved manually were all part of the Coq formalization described in Section 6.6, or can be deduced from the results in this formalization, so it is natural to wonder whether the Coq proofs of the Why3 results could have been based on that Coq formalization. We did not try, so the answer is not clear, but since we prove in Why3 only a few lemmas and the direct proofs that we wrote

```
function wd : graph -> set vertex -> set vertex
axiom wd_spec : forall g v' u. mem u (wd g v')
  <-> exists u1 l1 u2 l2. u1 <> u2
      /\ is_v_path g v' u u1 l1 /\ is_v_path g v' u u2 l2
      /\ forall z. LM.mem z l1 -> LM.mem z l2 -> z = u
```

Figure 8.5 – Definition of the set of V'-weakly deciding nodes in Why3

```
lemma wd_disjoint : forall g v' v. mem v (wd g v')
                                   -> not (mem v v')

lemma wd_property_2 : forall g v1' v2'. subset v1' v2'
  -> subset (wd g v1') (union v2' (wd g v2'))

axiom wd_property_3 :
  forall g v'. is_empty (wd g (union v' (wd g v')))
```

Figure 8.6 – Some properties about `wd`

in Coq are not so long, connecting both developments would probably not have been worth the effort.

Figure 8.6 shows a few results about `wd`. Lemma `wd_disjoint` states that, given a subset of vertices V', vertices that are V'-weakly deciding are not in V'. This corresponds to Property 6.2. Lemma `wd_property_2` states that if two subsets of vertices V_1' and V_2' verify $V_1' \subseteq V_2'$, then the V_1'-weakly deciding nodes are either in V_2' or V_2'-weakly deciding. This is the Why3 version of Property 6.3. Lemma `wd_property_3` states that, given a subset of vertices V', there is no $(V' \cup \mathsf{WD}_G(V'))$-weakly deciding nodes. This is the encoding in Why3 of Property 6.4. One can observe that this last lemma is actually introduced with `axiom`. We considered that, given that the proof of Property 6.4 is quite complex and that the proof in Coq of the corresponding lemma called `lemma53` (see 6.22) is quite long, it would have been too costly to prove it again in the context of the Why3 formalization. We thus decided to turn it into an axiom.

Functions `propagate`, `confirm` and `main`. The three functions forming the algorithm are written and annotated using the objects defined in the first part of the formalization that has just been presented. These implementations are very similar to what has been presented in the previous chapter and this one. We illustrate this claim by two excerpts of the code of these functions.

```
let propagate (g : graph) (w : B.t) obs (u v : vertex)
  requires { forall z. LL.length (H.([]) obs z) <= 1 }
  requires { mem u (support g) }
  (* P_1 *)
  requires { forall z. H.([]) obs z = Cons v Nil <-> z = u }
  (* P_2 *)
  requires { mem u w.B.contents }
  ensures { forall z. LL.length (H.([]) obs z) <= 1 }
  (* Q_1 *)
  ensures { forall z. mem u (obs_g g z w.B.contents)
            -> H.([]) obs z = Cons v Nil }
  (* Q_2 *)
  ensures { forall z. not (mem u (obs_g g z w.B.contents))
            -> H.([]) obs z = H.([]) (old obs) z }
  (* Q_3 *)
  ensures { forall z. mem z result.B.contents
            <-> z <> u /\ mem u (obs_g g z w.B.contents)
                /\ H.([]) (old obs) z <> Nil
                /\ cardinal (succ g z) > 1  }
```

Figure 8.7 – Excerpt of the definition of `propagate`

Figure 8.7 shows the beginning of the definition of `propagate`. Let us compare it with its contract given as Algorithm 8.5. It has five arguments as expected: the graph `g`, the set `w`, the labeling `obs`, and two vertices `u` and `v`. As for the preconditions and the postconditions, they are also very similar. All the preconditions ($(\mathbf{P_1})$ and $(\mathbf{P_2})$) and postconditions ($(\mathbf{Q_1})$, $(\mathbf{Q_2})$ and $(\mathbf{Q_3})$) given in Algorithm 8.5 are in the Why3 version. The Why3 version contains two preconditions and one postcondition more, but they are needed because of the encoding used in the formalization. They are explained in Section 8.5.3.

Another figure that illustrates how similar the Why3 development is to what is described in this thesis is Figure 8.8, that shows the definition of `main` in the Why3 development deprived of its annotations. We can connect this definition to the one given in Algorithm 7.3. Again, we ignore some of the particularities of the Why3 version. Some of them are explained in Section 8.5.3. Without going into details, we can observe they both have the same structure:

- The first step is the initialization of the main variables. W, *obs* and L are initialized on lines 2–4 of Algorithm 7.3; `w`, `obs` and `worklist` are initialized on lines 2–12 in Figure 8.8. The initialization is much longer in Why3. This is because we do not have at our disposal in Why3 instructions as concise as

```
1  let main g v' =
2    let w = B.empty () in
3    let obs = H.create 97 in
4    let worklist = B.empty () in

6    let tmp = impset_of_set v' in
7    while not (B.is_empty tmp) do
8      let u = B.choose_and_remove tmp in
9      B.add u w;
10     H.add obs u u;
11     B.add u worklist
12   done;

14   while not (B.is_empty worklist) do
15     let u = B.choose_and_remove worklist in
16     let candidates = propagate g w obs u u in

18     let new_nodes = B.empty () in
19     while not (B.is_empty candidates) do
20       let v = B.choose_and_remove candidates in
21       if confirm g obs v u then B.add v new_nodes
22     done;

24     while not (B.is_empty new_nodes) do
25       let v = B.choose_and_remove new_nodes in
26       B.add v w;
27       H.replace obs v v;
28       B.add v worklist;
29     done
30   done;

32   (w, obs)
```

Figure 8.8 – Definition of `main` in the Why3 development, with annotations removed

the ones we use in the pseudo-code version.

- The second step is a loop that iterates until all vertices in the worklist are processed. This loop is on line 5 of Algorithm 7.3 and on line 14 of Figure 8.8. The body of the loop is divided into three steps:
  - The first step is the call to `propagate` (line 8 of Algorithm 7.3 and line 16 of Figure 8.8).
  - The second step is the filtering of the candidate nodes returned by `propagate` using `confirm`. This corresponds to lines 9–15 of Algorithm 7.3 and lines 18–22 of Figure 8.8.
  - The third step is the update of the three main variables. This is the purpose of lines 16–18 of Algorithm 7.3 and lines 24–29 of Figure 8.8. Like for the initialization, we cannot write the Why3 version as concisely as the pseudo-code one.

8.5.2 Proof Effort

In this section, we focus on the proof effort needed to validate all the lemmas and annotations. This does not measure the effort needed to write the specifications and the annotations of the functions. Instead, this consists in determining by which backend solver each subgoal is proved, how much time it takes, whether this solver is automatic or not, and in case it is not how long is the manual proof. This help answering the question whether we really take advantage of the automation offered by Why3.

The backend solvers that we used are the following:

- four automatic SMT solvers, applied in that order:
  - Alt-Ergo (version 1.30),
  - CVC4 (version 1.5),
  - the E theorem prover (version 2.0),
  - Z3 (version 4.5.0);
- one interactive prover: Coq (version 8.6.1).

We give the results for each of the two parts of the Why3 development: the set of definitions and the algorithm itself.

Results for the preliminary definitions. The results are given in Figure 8.9. Among 34 subgoals, 10 are proved with Alt-Ergo, 14 are proved by CVC4, 4 are

	Alt-Ergo (1.30)	CVC4 (1.5)	Coq (8.6.1)	E (2.0)	Z3 (4.5.0)
Number	10	14	4	6	0
Min time (s)	0	0.02	0.27	0.01	0
Max time (s)	0.01	0.67	0.37	0.44	0
Avg time (s)	0.01	0.083	0.3	0.093	N/A

Figure 8.9 – Proof results for the preliminary definitions

	Alt-Ergo (1.30)	CVC4 (1.5)	Coq (8.6.1)	E (2.0)	Z3 (4.5.0)
Number	233	12	4	4	2
Min time (s)	0.01	0.08	0.32	0.08	0.34
Max time (s)	3.96	0.83	0.76	2.35	3.18
Avg time (s)	0.18	0.46	0.48	0.72	1.76

Figure 8.10 – Proof results for the algorithm

proved manually in Coq, 6 are proved using E and none are proved with Z3. The automatic proofs are all straightforward, since the maximal time needed is 0.67s. The manual proofs are together 100 lines of Coq proof long.

Note that these results do not take into account axiom `wd_property_3` (cf. Figure 8.6) since we chose to admit it.

Results for the algorithm. The results for the proofs of the correctness of `propagate`, `confirm` and `main` are given in Figure 8.10. There are much more goals than in the first part (255). The very large majority of them are proved automatically, especially by Alt-Ergo (233), but also by CVC4 (12), E (4) and Z3 (2). The remaining goals (4) are proved manually in Coq.

Some of the automatic proofs are more costly than those of the first part, but they are still reasonably fast (3.96s for the longest automatic proof). And most of them are really short. For instance, the average time of the proofs in Alt-Ergo is 0.18s.

Since, as explained in Section 8.5.1, the proofs done in Coq are those of the main invariants, one can expect quite complex, and thus long, proofs. This is indeed the case, since the four proofs together are 220 lines long. The longest one, the proof of preservation of invariant ($\mathbf{I_7}$) (cf. Algorithm 8.8 and Figure 8.2), is 160 lines long.

8.5.3 Particularities of the Why3 Formalization

The Why3 version is not identical to what has been described in the previous chapter and this one, since we need to adapt to the language and the libraries of

Why3. We present some of the differences between the two versions.

Ghost code. The Why3 formalization heavily relies on ghost code. This is code that is interleaved with the real code, but is not really executed. Instead, it just helps the verification. For example, Figure 8.11 shows the implementation of `confirm` in Why3. The set of children to process is assigned to `s` on line 9, and nodes are removed from `s` one by one on line 22. To keep track of the nodes that have already been processed, instead of describing them as the vertices in `succ g u` but not in `s`, we preferred to explicitly introduce a second set `s'` containing such vertices (line 10). Since we do not want to make any use of this set apart from helping verification, we make it ghost. To keep `s'` up-to-date, we add to it each node that is removed from `s` using a ghost instruction (line 23). The invariants of the loop on line 11 describe the relation between `s` and `s'`: their union is equal to the set `succ g u` (line 12), and they are disjoint (line 13).

Imperative sets. The sets manipulated in the algorithm are modified in-place. To implement them in Why3, we chose a library of so-called "imperative sets" with the following operations:

- a function called `choose_and_remove` which, as its name suggests, selects an element in an imperative set, removes it from the set and returns it. This is used to implement function `choose` presented in Section 7.3. Actually, a function called `choose` also exists in the library, but since each time we select a node in a set, we also remove it from that set (lines 5–6 and 9–10 of `propagate` (cf. Algorithm 7.1), lines 5–6 of `confirm` (cf. Algorithm 7.2), and lines 6–7 and 11–12 of `main` (cf. Algorithm 7.3)), this was natural to prefer `choose_and_remove`. An example call to `choose_and_remove` is line 22 of `confirm` in Figure 8.11.
- a function called `add` that adds an element to an imperative set (for instance, line 23 in Figure 8.11);
- a function called `remove` that removes an element from an imperative set;
- a function called `empty` that creates a new empty imperative set (for instance, line 10 of Figure 8.11);
- a function called `is_empty` that tests if an imperative set is empty (for instance, line 11 in Figure 8.11);
- A function called `mem` that tests the membership of an element in a set.

These imperative sets are of a different type than normal sets. To make the bridge between the two types, we have two operations at our disposal:

```
1  let confirm g obs u u'
2      requires { forall z. LL.length (H.([]) obs z) <= 1 }
3      ensures { result <->
4                  exists v v'. mem v (succ g u)
5                  /\ H.([]) obs v = Cons v' Nil
6                  /\ v' <> u' }
7  =
8    let res = ref false in
9    let s = impset_of_set (succ g u) in
10   let ghost s' = B.empty () in
11   while not (B.is_empty s) do
12     invariant { union s.B.contents s'.B.contents == succ g u }
13     invariant { inter s.B.contents s'.B.contents == empty }
14     invariant { !res <->
15                   exists v v'.
16                   mem v (diff (succ g u) s.B.contents)
17                   /\ mem v (succ g u)
18                   /\ H.([]) obs v = Cons v' Nil
19                   /\ v' <> u' }
20     invariant { subset (s.B.contents) (succ g u) }
21     variant { cardinal s.B.contents }
22     let v = B.choose_and_remove s in
23     ghost B.add v s';
24     if H.mem obs v && not (eq_vertex u' (H.find obs v)) then
25       res := true;
26   done;
27   !res
```

Figure 8.11 – Implementation of `confirm` in Why3, with its annotations

- The first one is a function called `impset_of_set` that converts a set into an imperative set. This function is not provided by the library, but was written by us. It is used for example in `confirm` (line 9 of Figure 8.11) to convert the set of children into the imperative set `s`.
- The second one is available only in the logical world. Imperative sets are equipped with a logical field called `contents` that retrieves their contents as normal sets. It is used for example in a loop invariant in function `confirm` (line 20 in Figure 8.11).

Labeling as a hash table. To implement the labeling *obs*, we chose a library of hash tables. The difficulty with this implementation is that the kind of hash tables we use are allowed to associate multiple values to the same key. Actually, only the most recently added value can be seen, but if this value is removed, then the second most recently added value becomes visible. This explains the logical representation of these hash tables as maps from keys to lists of values. In our implementation, we take care of always having at most one value associated to each key. For example in the update step of function `main`, we use function `replace` and not `add` (line 27 in Figure 8.8), since the vertex whose label is updated has already a label (the one that has just been propagated). But we need to keep track of this invariant in the annotations. For instance, the Why3 implementation of `confirm` has a precondition (cf. Figure 8.11), while our pseudo-code version does not (cf. Algorithm 8.7). This precondition ensures that the received labeling associates at most one value to each key, i.e. one label to each node. In the contract of `propagate` given in Figure 8.7, we specify this property both as a precondition and as a postcondition. This means that this property is preserved by function `propagate`.

Function `support`. Our choice to model graphs using a function `support` returning the set of nodes in a graph (cf. Section 8.5.1) make some annotations heavy, since we need to specify multiple times that all the manipulated nodes are nodes in the graph `g`, i.e. are in `support g`. For instance, `propagate` gets as a precondition that the input vertex `u` is a valid vertex of the input graph `g` (cf. Figure 8.7). Since this property seems trivial, one could be tempted to remove all these annotations, but in this case, another modeling of the finiteness of the graph has to be introduced.

8.5.4 Extraction into OCaml

Why3 has an extraction mechanism similar to Coq that allows to produce a correct-by-construction executable program, e.g. in OCaml. However, we did not have

time to understand and use this mechanism. Instead, we chose the simpler path of implementing the algorithm manually in OCaml (see Chapter 9). The drawback of this approach is a weaker confidence in the correctness of the OCaml implementation, but since WhyML and OCaml are really similar languages, we can still have a relatively good confidence in the OCaml implementation, since it resembles the Why3 version.

In this chapter, we saw that the algorithm that we propose is indeed correct. The mechanized proof in Why3 gives us a high confidence in its correctness. We still need to justify the second claim that we made, i.e. that our algorithm is faster than Danicic's. This is the purpose of the next chapter.

Chapter 9

Experimental Comparison of Danicic's and the New Algorithms

In this chapter, we compare experimentally the so-called optimized algorithm presented in Chapter 7 and Danicic's algorithm presented in Chapter 6 to measure experimentally whether our proposition is really optimized with respect to the latter.

Actually, we do not have only two implementations to test. Indeed, we also want to test variants of each algorithm. For example, we want to check if the Coq extraction of Danicic's algorithm mentioned in Section 6.6.1 has reasonable performance. We also want to check if the two optimizations described in Section 6.5.2 improve the running time of the algorithm and measure how much.

This chapter is organized as follows. Section 9.1 presents the framework we used to perform these experiments. Next, Section 9.2 presents the variants of the two algorithms that we have tested. Section 9.3 presents the results obtained and tries to interpret them.

9.1 Methodology

9.1.1 Remarks about the Implementations

This section presents the common features that are shared by all the implementations we have tested.

The first point that we want to highlight is that all the implementations that we have executed are written in the same language: OCaml. The only odd case is the Coq extraction. Indeed, since it is originally written in Coq, we could argue

that, contrary to the other implementations, it is not written in OCaml. But since the version that we finally execute is the extraction into OCaml, we can also consider that the source code compiled and executed is OCaml code. Admittedly, since the extraction into OCaml produces more complex code than what we would write directly in OCaml, this penalizes the Coq extraction with respect to the other implementations. This may explain why the Coq extraction is so slow, as discussed in Section 9.3.1.

All hand-written OCaml implementations, i.e. all but the Coq extraction, are written using the same generic graph library called OCamlgraph (version 1.8.8) [CFS07]. This library provides convenient functions to access the parents or the children of a given node and to traverse the whole graph. It also comes with high-level operations, such as:

- a function `copy` that clones a given graph;
- a function `check_path` that tests the reachability of a node from another node and that comes with caching, so that new calls to the same function can take advantage of previously computed results;
- helper functions to write in a simple way dataflow analyses.

This library allows to write the implementations in a concise way, which make them relatively close to the high-level description given in this thesis, and thus easy to write and to understand. OCamlgraph allows to choose between multiple graph implementations:

- imperative, i.e. mutable, or persistent, i.e. immutable;
- directed or undirected;
- standard (every vertex can easily access its successors, but not its predecessors) or bidirectional (every vertex can easily access both its successors and its predecessors).

Initially, we used a standard directed imperative implementation.[1] But since both algorithms are based on backward traversal, we switched to a bidirectional directed imperative implementation, that takes more memory space, but in which accesses to predecessors are claimed to be in constant time instead of being linearly dependent on the size of the graph.

Another fact that we had to take into account is that Danicic's algorithm and the optimized one do not compute the same set. Indeed, as explained in Chapter 7,

[1] The experimental results of [LKL18c] were obtained using that graph representation.

from an initial set V' of vertices of graph G, Danicic's algorithm computes the weak control-closure of V' in G, $\mathsf{WCC}_G(V')$, while the optimized algorithm computes only $V' \cup \mathsf{WD}_G(V')$, i.e. the addition of all the V'-weakly deciding nodes to V'. Recall that the difference is that $V' \cup \mathsf{WD}_G(V')$ contains by definition all the V'-weakly deciding vertices, while $\mathsf{WCC}_G(V')$ contains only the V'-weakly deciding nodes that are reachable from V'. Formally:

$$\mathsf{WCC}_G(V') = V' \cup (\mathsf{WD}_G(V') \cap \mathsf{R}_G(V'))$$

This is not clear whether this difference favors or handicaps Danicic's algorithm with respect to the optimized version. Indeed, $\mathsf{WCC}_G(V')$ is a smaller set than $V' \cup \mathsf{WD}_G(V')$, so it may be easier to compute. On the other hand, it is in general easier to detect weakly deciding nodes with respect to a larger set, since paths that lead to that set are shorter. Moreover, if we compute $V' \cup \mathsf{WD}_G(V')$, we do not need the reachability tests that Danicic's algorithm realizes on each considered vertex. This last point is discussed in Section 9.3.5.

To make the comparison fairer, we ensure that all the implementations that we test compute the same result. In order to do this, we add to the algorithms that do not compute the weak control-closure directly a filtering step at the end of the algorithm that preserves only the vertices reachable from the initial subset V', as suggested in Section 7.1.

9.1.2 Description of the Testing Procedure

This section presents the methodology we adopted for the experimentation.

The implementations are run on graphs that are randomly generated using OCamlgraph. More precisely, we use the function `Rand.graph` of OCamlgraph that takes as parameters a number v of vertices and a number e of edges and generates a random graph with v vertices and e edges. For all the experiments, we set the number of edges to twice the number of vertices. The initial set V' consists in three vertices randomly taken in the graph. These choices are not made to get realistic graphs, but to get cases where computing weak control-closure is often not trivial, i.e. cases where the weak control-closure is often not reduced to the initial subset V'. We informally confirmed that fact during our experiments, since in most of the cases the weak control-closure is not reduced to the initial V'. Moreover, in these cases, the resulting closure nearly always represents a significant part of the set of vertices of the graph. The exact choice of 2 as the ratio between the number of vertices and the number of edges is discussed in Section 9.3.8.

Before even running the first experiments, we wanted to obtain some guarantees that our implementations are correct. Indeed, among our implementations, only the Coq extraction is certified. For that, we ran all of them on random small graphs

(typically 30 nodes, and thus 60 edges) and checked that they all computed the same weak control-closure, and in particular the same results as the certified Coq extraction. Moreover, during the experiments, when multiple implementations are run on the same graph, we systematically check that the computed results are identical. This does not prove that all the implementations are correct, but greatly increase our confidence that they are.

The experimentation consisted in running each implementation on hundreds of random graphs. The parameters used vary on the implementations. For expensive implementations, such as the Coq extraction, the graphs contain a few hundreds of nodes and the step between each graph size tested is 10. For the most efficient implementations, the graphs contain up to hundreds of thousands of nodes and the step used is 10 000. The exact parameters used are given in Section 9.3. For each implementation and each size of graph tested, we run the implementation on 10 graphs of this size. As discussed above, for the majority of the 10 graphs, the weak control-closure is rather large. We decided to ignore the cases in which the weak control-closure is equal to the initial V' and thus contains only three vertices. Indeed, in these cases, the execution time is insignificant. The result for that implementation and that size of graph is the average of the running times in the cases where the weak control-closure is not trivial.

In terms of hardware, experiments have been performed on an Intel Core i7 4810MQ with 8 cores at 2.80 GHz and 16 GB RAM.

9.2 Presentation of the Implementations

In this section, we list the implementations that we have tested in the experimentation. First, we present the variants of Danicic's algorithm in Section 9.2.1. Then we present the variants of the optimized algorithm in Section 9.2.2.

9.2.1 Variants of Danicic's Algorithm

Let us first recall the two optimizations that we proposed in Section 6.5.2.

- The first optimization consists in detecting as many critical edges as possible during each iteration. This is denoted as optimization 1.
- The second one consists in weakening the definition of critical edge. This is denoted as optimization 2.

From a given implementation of Danicic's algorithm, we can thus derive four variants depending whether none, only one or both optimizations are selected.

In all the implementations of Danicic's algorithm, including the Coq extraction, the computation of the observable vertices is performed as described in [DBH+11]. Given a graph G, a subset of vertices V' and a node u in G, the set of observable vertices from u in V' is computed by removing from G all the outgoing edges from nodes in V' and selecting all the nodes in V' that are reachable from u in the resulting graph H. Removing the edges that have their sources in V' guarantees that the nodes in V' that are reachable from u in H are first-reachable from u in V' in graph G, i.e. are observable from u in V' in graph G. In addition to the Coq extraction, we propose two variants of Danicic's algorithm that follow this scheme, but differ in the implementation details.

The Coq extraction. The first implementation of Danicic's algorithm that we consider is the Coq extraction mentioned in Section 6.6.1. Recall that it is the extraction of the implementation in Coq of Danicic's algorithm based on a prototype graph library presented in [DERV15]. As mentioned in Section 6.6.1, it implements optimization 1 but not optimization 2. It is really naive in the sense that little information is shared between operations, even inside a given iteration. Consider a given iteration in the algorithm and let W be the auxiliary set used for the algorithm. Testing if an edge (u, v) is W-critical is realized in three steps.

- We compute the graph H from the graph G by removing all the outgoing edges starting in W. Then, we compute in H the set of nodes in W reachable from v. The vertices that we get are the observable vertices from v in W in graph G. If this set does not contain exactly one element, (u, v) is not W-critical. Otherwise, we start the second step.

- We compute the set of observable vertices of node u in W in graph G. For that, we reuse the graph H computed in the first step and compute the set of vertices in W reachable from u in H. If there is strictly less than two elements observable from u in W, (u, v) is not W-critical. Otherwise, we start the third step.

- We check if u is reachable from V' in G. This requires a traversal of G. If u is reachable from V', then (u, v) is W-critical. Otherwise, it is not.

Since optimization 1 is implemented, whether (u, v) is W-critical or not, we then process one by one all the edges that have not been processed yet in this iteration. For that, we apply the same three steps that we have just described.

Note how naive this implementation is. In particular, in the calculation of the observable set of u, we do not use the observable set of v that has just been computed, while all the nodes observable from v are straightforwardly observable from u if u is not in W. Moreover, when we analyze a new edge, we do not use any

of the intermediate results obtained when analyzing the previous edges in the same iteration. In particular, H is recomputed, whereas it has not changed, and a new traversal of G is performed for the reachability test that does not take advantage of the reachability tests performed for the previous edges.

As discussed in Chapter 7, the main weakness of Danicic's algorithm is the absence of propagation of information between iterations. The Coq extraction, as any implementation of Danicic's algorithm, inherits this weakness. What makes the Coq extraction particularly naive is that it does not propagate intermediate results even inside each iteration, apart from the graph H that is shared between the calculations of the observable sets of u and v in the analysis of edge (u, v). Its only smart aspect is the implementation of optimization 1. But it is not sure that this optimization is really interesting in this case. Indeed, the goal of optimization 1 is to reduce the number of iterations and do more work in each iteration. It is interesting if doing some operations in a new iteration is more costly than doing them in the same iteration. Considering the naive aspects of the Coq extraction, it is not clear that this is the case, and thus it is not clear that this optimization has a noticeable impact on the performance of the Coq extraction.

Naive OCaml implementation. We implemented Danicic's algorithm as naively as the Coq extraction, but in OCaml and using OCamlgraph. Actually, this OCaml implementation is even more naive than the Coq extraction, since it does not implement any of the two optimizations and does not even share the graph H between the calculations of the observable sets of u and v in the analysis of edge (u, v).

There are four variants of this naive implementation of Danicic's algorithm: without the two optimizations, with optimization 1, with optimization 2, and with both. As in the case of the Coq extraction, we expect the first optimization to be of little interest, since there is no sharing even inside an iteration. Contrary to the Coq extraction, we have two versions, one without optimization 1 and one with optimization 1, which allows to confirm or refute that intuition experimentally (see Section 9.3.2).

Smart OCaml implementation. We implemented Danicic's algorithm much more smartly, taking advantage of caching in several places. First, we compute once at the beginning of the algorithm the set of nodes reachable from V' and use this set for every reachability check needed later in the algorithm. Moreover, at the beginning of an iteration, we create the graph H from G by removing the outgoing edges starting from W and we compute simultaneously for every node in G its observable set in W using the helper functions provided by OCamlgraph to write a dataflow analysis. This analysis computes the observable set of a node from the observable sets of its children, which allows to compute the observable information for the whole graph G rather efficiently. Actually, this is close to what

is described in the example of Figure 6.17, where each iteration annotates every node with its set of observables (cf. Figures 6.17b, 6.17d and 6.17f).

Again, we have four variants: without the two optimizations, with optimization 1, with optimization 2, and with both. We also have a fifth variant that implements both optimizations, but does not check during the algorithm if the tested nodes are reachable from V'. Instead, it implements the filtering step discussed in Section 9.1.1 that removes all the unreachable nodes at the end of the algorithm. This fifth implementation allows to check whether the reachability tests performed during Danicic's algorithm have a negative impact on the performance of the algorithm (see Section 9.3.5).

9.2.2 Variants of the Optimized Algorithm

We have implemented and tested three variants of the optimized algorithm.

Why3 implementation. As explained in Section 8.5.4, we did not extract the WhyML code into OCaml using the extraction mechanism of Why3. However, since WhyML is similar to OCaml, we adapted it manually. We took the WhyML code, removed the annotations, replaced the graph library used in Why3 with calls to OCamlgraph, and adapted the few remaining parts that needed to be adapted to OCaml. This gives an implementation of the optimized algorithm that is really close to the Why3 formalization.

OCaml implementation. We implemented the optimized algorithm directly in OCaml, using OCaml idioms and the full power of OCamlgraph.

OCaml implementation with two traversals. The optimized algorithm described in Chapter 7 adds the V'-weakly deciding candidates it detects during the propagation to a set that is filtered afterwards, so that only true V'-weakly deciding nodes are preserved. Since a lot of nodes are added to the set (all the nodes whose label changes during the propagation and that have at least two children), the manipulation of this set may be costly. As discussed in Remark 7.2, instead of adding all the candidates to a set during the propagation, we could traverse the graph a second time to perform the filtering. We implemented this variant in order to check whether it is competitive with respect to the standard optimized algorithm (see Section 9.3.7).

9.3 Results

This section presents the main results of the experiments.

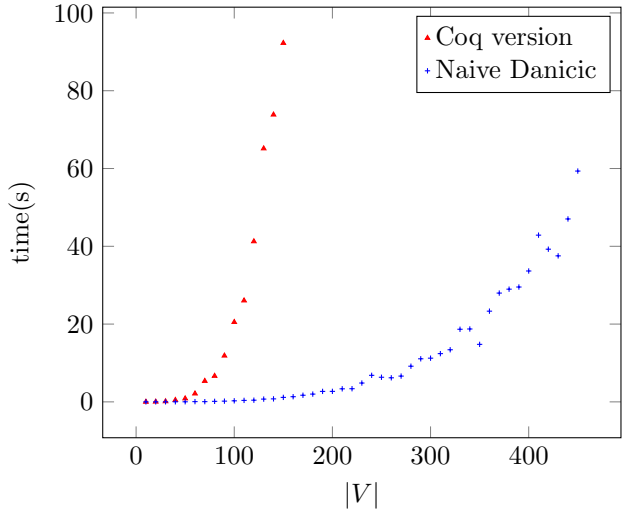


Figure 9.1 – Comparison between the Coq extraction and a naive implementation in OCaml of Danicic's algorithm

Implementations are compared pairwise. In each diagram, the ordinate is the execution time in seconds and the abscissa is the number v of nodes in the graph. Recall the the number of edges e is equal to $2 \times v$, except in Section 9.3.8. For the sake of clarity, we adopt the following convention. In each diagram, the implementation that is expected to be slower is represented using red triangles, while the implementation that is expected to be faster is represented using blue plus signs.

9.3.1 Comparison between the Coq Extraction and a Naive Danicic Implementation

The first diagram that we present compares the Coq extraction and the naive implementation of Danicic's algorithm without any optimization. As discussed in Section 9.2.1, this implementation of Danicic's algorithm is more naive than the Coq extraction, since it does not implement any of the two optimizations, while the Coq extraction implements optimization 1. However, the Coq extraction is written in Coq using a prototype graph library, which has probably a more negative impact than optimization 1 has a positive impact on the performance.

Figure 9.1 shows the results. The Coq extraction was run on graphs of sizes equal to multiples of 10 between 10 and 150. The naive Danicic implementation was run on graphs of sizes equal to multiples of 10 between 10 and 450.

We observe that the Coq extraction explodes for barely more than 100 nodes, while the OCaml implementation can handle graphs with a few hundreds of nodes. The Coq extraction is thus clearly penalized by the fact that it is written in Coq

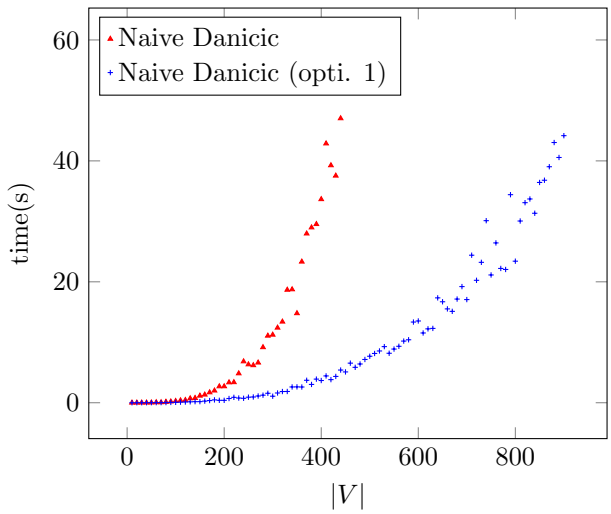


Figure 9.2 – Comparison between two naive OCaml implementations of Danicic's algorithm, with and without optimization 1

and by the prototype graph library it relies on.

9.3.2 Impact of the Optimizations on the Naive Danicic Implementation

We study the impact of optimization 1 on the naive OCaml implementation of Danicic's algorithm by comparing its implementations with and without optimization 1. As discussed in Section 9.2.1, we expect the optimization to have little impact on the performance of the algorithm.

The comparison is shown in Figure 9.2. The naive Danicic implementation was run on graphs of sizes equal to multiples of 10 between 10 and 450, while the naive Danicic implementation with optimization 1 was run on graphs of sizes equal to multiples of 10 between 10 and 900.

Rather surprisingly, optimization 1 has a noticeable impact on the performance of the algorithm, since the implementation with optimization 1 can handle graphs that are twice as large as those handled by the implementation without optimization 1 in the same time. What can explain this fact is that starting over repeatedly a new iteration implies that the same edges are analyzed again and again at the beginning of each iteration. This does not happen when optimization 1 is implemented.

We now study the impact of optimization 2, alone or in combination with optimization 1 on the naive OCaml implementation. Alone, optimization 2 does

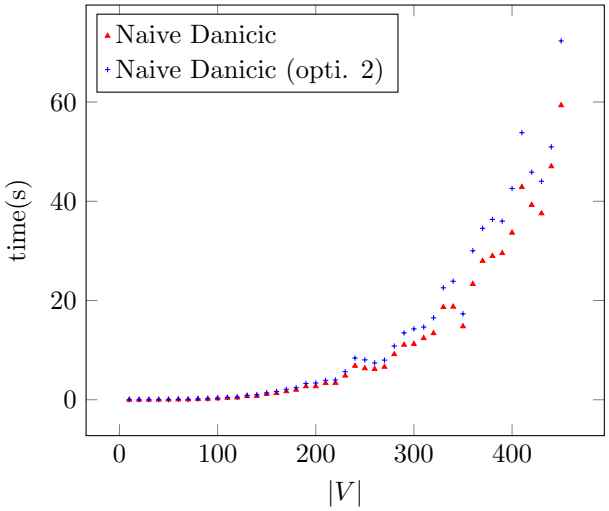


Figure 9.3 – Comparison between two naive OCaml implementations of Danicic's algorithm, with and without optimization 2

not seem like a great improvement, since its impact is mainly to change the order in which the nodes are detected. However, in combination with optimization 1, it allows to detect more nodes at each iteration, and thus probably to reduce the number of iterations.

The results are given in Figure 9.3 for optimization 2 alone and Figure 9.4 for optimization 2 with optimization 1. In Figure 9.3, both algorithms were run of graphs of sizes equal to multiples of 10 between 10 and 450. In Figure 9.4, both algorithms were run of graphs of sizes equal to multiples of 10 between 10 and 900.

Surprisingly, alone or in combination with optimization 1, optimization 2 deteriorates the performance of the algorithm. In both cases, the negative impact of optimization 2 is noticeable, but it seems more important in combination with optimization 1. We may explain this negative impact by the fact that, since the definition of critical edge is weaker, more edges look like critical edges and thus more computation is needed to refute that they are critical. The fact that the performance does not improve indicates that these additional calculations are not counterbalanced by an early detection of weakly deciding nodes. This may mean that, at least in the random graphs that we have generated, few edges are critical in the weak sense but not critical in the standard sense.

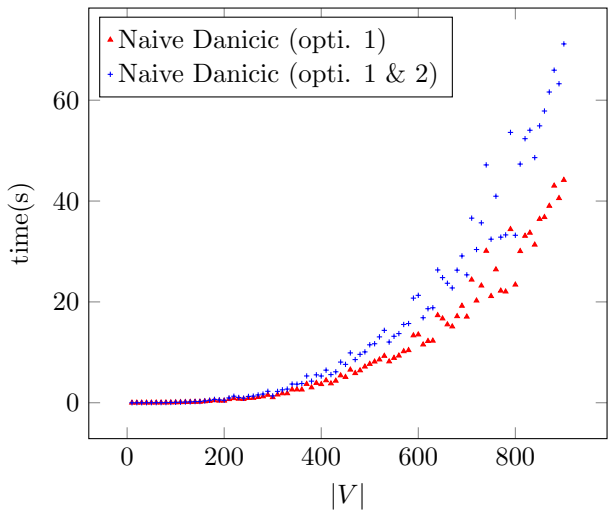


Figure 9.4 – Comparison between two naive OCaml implementations of Danicic's algorithm, one with optimization 1 and one with optimizations 1 and 2

9.3.3 Comparison of Naive and Smart Danicic Implementations

We now compare the naive and smart OCaml implementations of Danicic's algorithm without any optimization. Recall the the naive optimization does a lot of redundant computation, while the smart one caches most of the results and reuses them directly. We expect an important speedup in the smart Danicic implementation.

Figure 9.5 shows the results. The naive implementation was run on graphs of sizes equal to multiples of 10 between 10 and 450. The smart implementation was run on graphs of sizes equal to multiples of 10 between 10 and 1 600.

Figure 9.5 clearly shows that the smart implementation outperforms the naive implementation. While the naive implementation explodes for graphs with 500 nodes, the smart one can handle graphs that are three times larger.

9.3.4 Impact of the Optimizations on the Smart Danicic Implementation

As in the case of the naive implementation, we study the impact of optimizations 1 and 2 on the performance of the smart implementation. Since the smart implementation shares intermediate results in each iteration, starting a new iteration should be costly in comparison to performing more work in the same iteration. Thus optimization 1 should be really interesting. As in the case of the naive im-

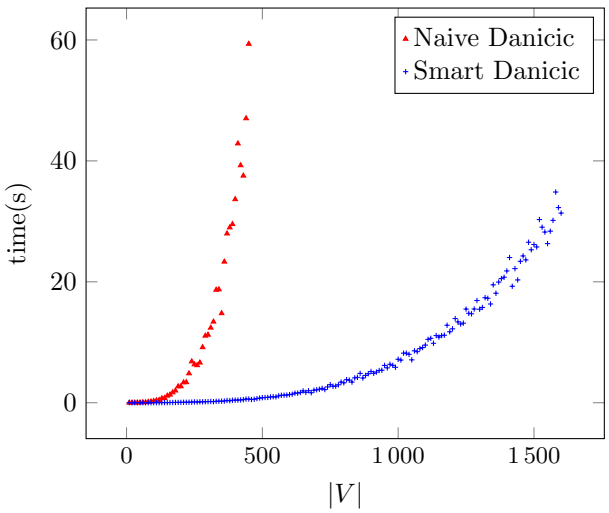


Figure 9.5 – Comparison between the naive and smart OCaml implementations of Danicic's algorithm

plementation, optimization 2 is expected to have no impact alone, and to slightly improve the performance in combination with optimization 1.

Figure 9.6 shows the comparison of the smart implementation without any optimization and the smart implementation with optimization 1. The implementation without any optimization was run on graphs of sizes equal to multiples of 10 between 10 and and 1 600. The implementation with optimization 1 was run on graphs of sizes equal to multiples of 100 between 100 and 10 000.

As expected, optimization 1 significantly improves the smart implementation. The optimized variant can handle graphs with 10 000 nodes that are five times larger than the graphs on which the unoptimized variant reaches its limits.

Figure 9.7 and Figure 9.8 shows the impacts of optimization 2, without and with optimization 1 respectively. In Figure 9.7, both algorithms were run of graphs of sizes equal to multiples of 10 between 10 and 1 600. In Figure 9.8, both algorithms were run of graphs of sizes equal to multiples of 100 between 100 and 10 000.

Using optimization 2 alone does not reduce the execution times of the implementations, but, contrary to the naive implementation, it does not increase them either. Using optimization 2 in combination with optimization 1 does not have a significant impact, but it seems that, for large graphs of at least 6 000 nodes, the variant with both optimizations is slightly faster than the variant with only optimization 1. This can be explained as follows. As in the case of the naive implementation, the definition of critical edge is weaker, thus more edges look like critical edges and thus more computation is needed to refute that they are critical. But this time, the additional computation is insignificant. Indeed, in the case of

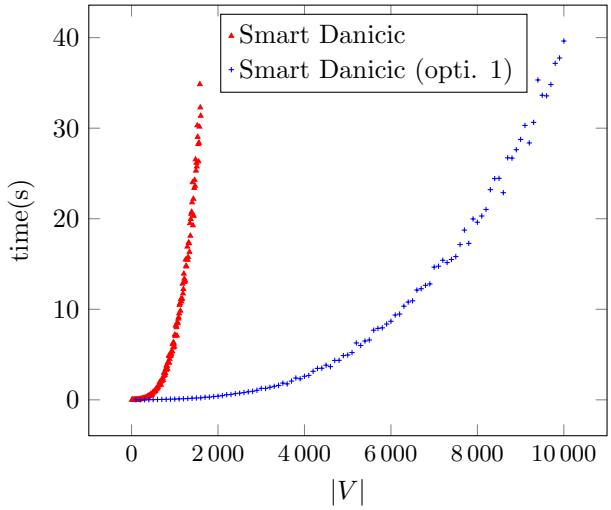


Figure 9.6 – Comparison between two smart OCaml implementations of Danicic's algorithm, with and without optimization 1

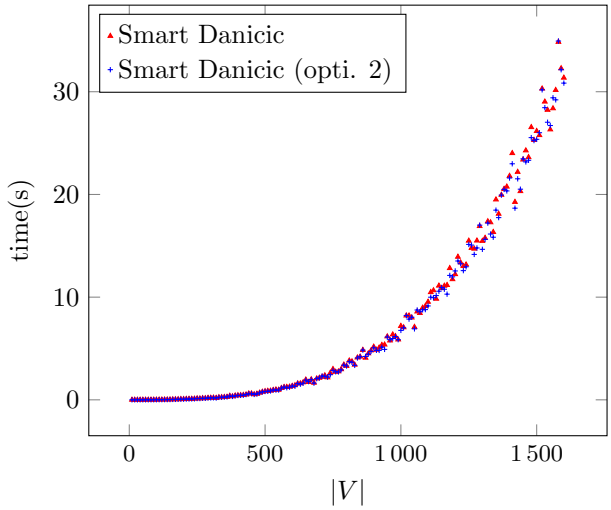


Figure 9.7 – Comparison between two smart OCaml implementations of Danicic's algorithm, with and without optimization 2

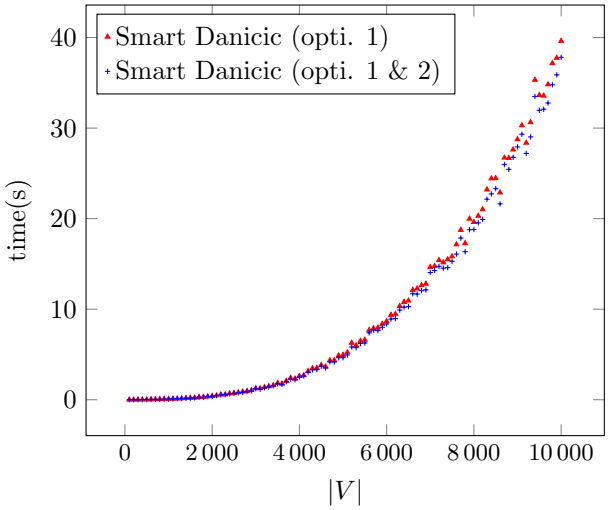


Figure 9.8 – Comparison between two smart OCaml implementations of Danicic's algorithm, one with optimization 1 and one with optimizations 1 and 2

the naive implementation, testing an edge means computing the observable sets of its endpoints. In the case of the smart implementation, the observable set of every node has already been computed at the beginning of the iteration. Testing an edge amounts to reading the precomputed annotations, this is negligible.

9.3.5 Impact of the Reachability Tests on a Smart Danicic Implementation

To analyze whether it is more interesting to test the reachability of the considered nodes during the algorithm like in standard Danicic's algorithm or at its end, we compare the smart implementation with both optimizations that we have just analyzed to a variant where the reachability checks are postponed until the end of the algorithm. This variant simply applies a filtering step at the end, similarly to what we perform for the optimized algorithm.

The results of the comparison are shown in Figure 9.9. Both implementations were run on graphs of sizes equal to multiples of 100 between 100 and 10 000.

The running times are similar for both variants. The variant with the filtering step at the end seems faster for large graphs of at least 7 000 nodes. This can be attributed to the smaller number of reachability tests performed, or seen as a sign that, as suggested in Section 9.1.1, detecting W-weakly deciding vertices is easier if W is larger.

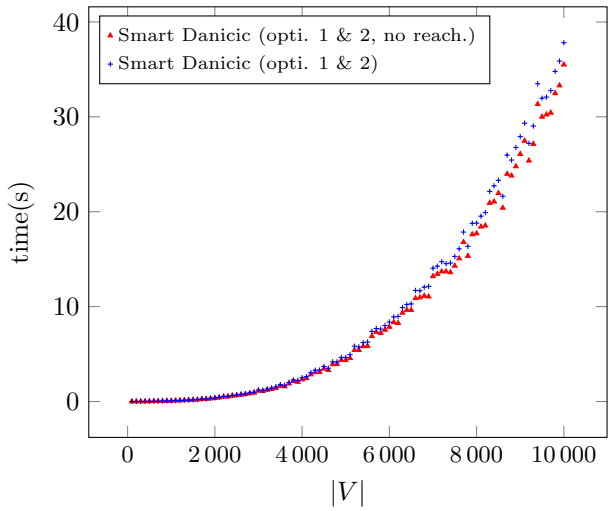


Figure 9.9 – Comparison between two smart OCaml implementations of Danicic's algorithm with both optimizations, with or without reachability checks

9.3.6 Comparison between Danicic's and the Optimized Algorithms

The comparison between an implementation of Danicic's algorithm and an implementation of the optimized algorithm is the main result of the experiments. We use the smart OCaml implementation with optimizations 1 and 2 as implementation of Danicic's algorithm, and the OCaml implementation as implementation of the optimized one.

Figure 9.10 shows the results. The implementation of Danicic's algorithm was run on graphs of sizes equal to multiples of 100 between 100 and 10 000, while the implementation of the optimized algorithm was run on graphs of sizes equal to multiples of 100 between 100 and 30 000.

As already discussed in Section 9.3.4, Danicic's algorithm with optimizations 1 and 2 can handle graphs with 10 000 nodes. But for the largest graphs, it takes 40 s to execute. There is no comparison with the optimized algorithm that takes less of 1 s even for graphs with 30 000 nodes.

9.3.7 Comparison between the Implementations of the Optimized Algorithm

We have just seen that the OCaml implementation of the optimized algorithm significantly outperforms one of our implementations of Danicic's algorithm. We now want to test how the implementations of the optimized algorithm compare

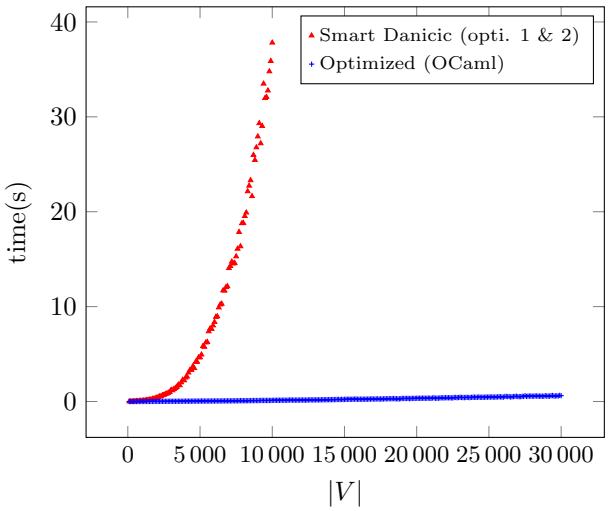


Figure 9.10 – Comparison between the smart OCaml implementation of Danicic's algorithm with both optimizations and the optimized algorithm

with each other.

We first compare the version that is close to the WhyML code with the OCaml implementation.

Figure 9.11 shows the results of the comparison. Both algorithms were run on graphs of sizes equal to multiples of 10 000 between 10 000 and 500 000.

Contrary to the Coq extraction that is much slower than all the other implementations, the manual extraction from Why3 is competitive with respect to the implementation in OCaml. More precisely, the OCaml implementation is faster, but only a little faster. On graphs with about 500 000 nodes, the OCaml version runs in around 30 s. The manual extraction is around 4 s slower.

The other implementation that we want to compare with the OCaml version is the implementation that performs a second traversal after the propagation instead of analyzing a set of candidates built during the propagation.

The comparison is shown in Figure 9.12. Both algorithms were run on graphs of sizes equal to multiples of 10 000 between 10 000 and 500 000.

Like the manual extraction, the implementation with a second traversal is slower than the standard OCaml implementation, but reasonably slower. This experimentally confirms that preserving during the propagation all the nodes whose labels change and that have at least two children is too simple to improve significantly the running time of the algorithm.

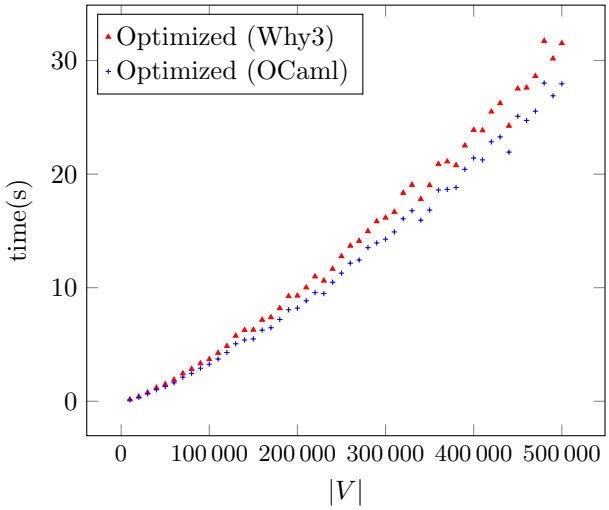


Figure 9.11 – Comparison between the manual extraction from WhyML and the OCaml implementation of the optimized algorithm

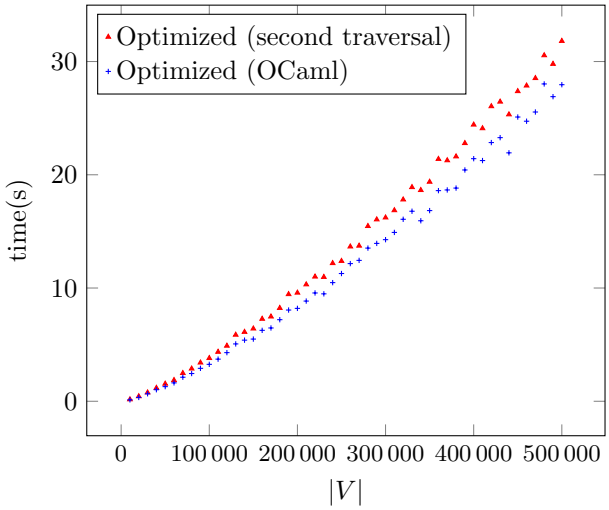


Figure 9.12 – Comparison between the implementation with a second traversal and the standard implementation of the optimized algorithm

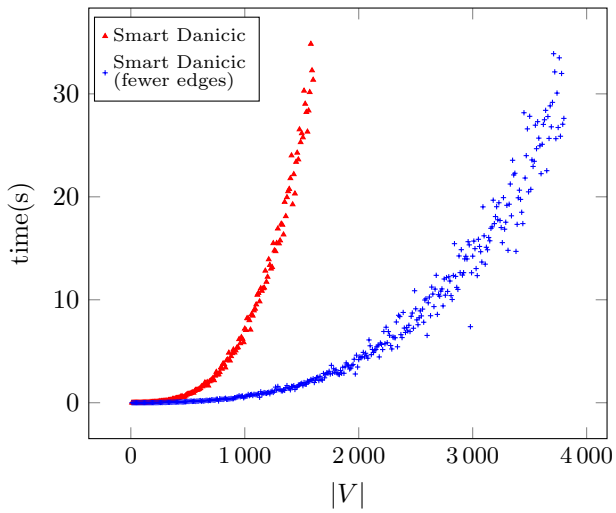


Figure 9.13 – Comparison of the smart implementation of Danicic's algorithm on graphs with $e/v = 2$ and $e/v = 1.5$

9.3.8 Impact of the Number of Edges on the Experiments

The last experiment that we present does not compare another implementation. It was performed to give confidence in the experimentation. Indeed, the graphs that are used in the experiments are randomly generated by OCamlgraph with twice as many edges as vertices. The results obtained in this setting may not be representative of the general case. For example, the random generator of OCamlgraph may introduce some bias in the results. We did not check whether it is the case. But at least we wanted to question the choice of 2 as the ratio between the number of edges e and the number of vertices v that seems rather arbitrary. Actually, it is somewhat arbitrary, since it has no precise meaning, but was chosen intuitively so that most of the generated graphs give a rather large weak control-closure. To check that the results are not completely different with fewer edges, we reproduced two experiments with the smaller ratio $e/v = 1.5$.

Figure 9.13 shows the results for the smart OCaml implementation of Danicic's algorithm. The smart implementation was run on graphs with $e/v = 2$ of sizes equal to multiples of 10 between 10 and 1 600, and on graphs with $e/v = 1.5$ of sizes equal to multiples of 10 between 10 and 3 800. Figure 9.14 shows the results for the OCaml implementation of the optimized algorithm. This implementation was run on graphs of sizes equal to multiples of 10 000 between 10 000 and 500 000, first with $e/v = 2$ and then with $e/v = 1.5$.

For both implementations, the running times are noticeably smaller for the graphs with fewer edges. For instance, regarding the smart OCaml implementation of Danicic's algorithm, graphs with around 4 000 nodes and a ratio $e/v = 1.5$ are

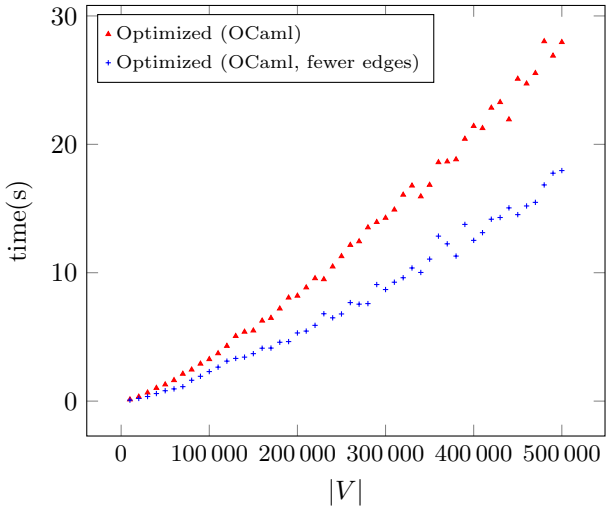


Figure 9.14 – Comparison of the OCaml implementation of the optimized algorithm on graphs with $e/v = 2$ and $e/v = 1.5$

processed in approximatively the same time as graphs with around 1 500 nodes and a ratio $e/v = 2$.

However, the trends that are revealed are rather similar for both values of e/v. Regarding the smart OCaml implementation of Danicic's algorithm, whether e/v is equal to 2 or 1.5, the implementation reaches its limits for graphs with a few thousand nodes. More importantly, in both cases, the implementation of the optimized algorithm is much more efficient than the implementation of Danicic's algorithm.

The main result of the experiments is the comparison of Danicic's and the optimized algorithms that confirms our expectations that the optimized algorithm is faster than Danicic's algorithm. The experimental results even show that the difference in terms of running times between both algorithms is substantial.

The experiments also give more surprising results. Optimization 1 of Danicic's algorithm that consists in detecting at each iteration all the critical edges instead of at most one is always an interesting optimization. On the contrary, optimization 2 that consists in relaxing the definition of critical edge is rarely interesting and can even deteriorate the performance of the algorithm. Regarding the optimized algorithm, building a set of candidate nodes during the propagation is beneficial, but only slightly.

Chapter 10

Conclusion

10.1 Summary

In this thesis, we have provided machine-checked solutions to two problems about program slicing: the justification of its use for runtime error detection and the design of an optimized algorithm for arbitrary control dependence.

The first problem is formalized in the Coq proof assistant using a simple representative imperative language that contains both errors, modeled as assertions, and non-termination, in the form of potentially non-termination loops. For this language, we show that classic program slicing based on standard control and data dependence relations can be used in a sound way for verification purposes if the results are interpreted with care. For that, we prove that slicing in this context verifies a weaker soundness property than in the classic case, but that this weakened property is strong enough to interpret the verification results obtained on a slice in the context of the initial program. Namely, if a slice is proved to be free from runtime errors, then the statements of that slice will not produce runtime errors in the original program either. On the contrary, if a statement in the slice can trigger an error, something can be deduced for the original program, but not necessarily that the same error can be triggered. There are three possible cases. The same error being triggered is one of them, but there are two other possibilities. The error can be hidden either by another error or by a non-terminating statement occurring before in the initial program. In the common case where the original program is expected to always terminate, these three cases are viewed as anomalies, so the detection of an error in the slice, even if it does not necessarily lead to the detection of the same error in the original program, is of great interest. These results justify the use of program slicing in verification techniques, where slicing is applied to simplify the error detection step. Moreover, a certified slicer can be extracted from the Coq development for the representative language that

we consider.

Such a slicer is of limited interest, since it slices only programs written in the simple language we use to model the first problem. To have more applicable results, it is natural to consider larger languages. But to factorize the effort and taking advantage of the language-independent nature of slicing, we do not aim for a slicer for a specific language but rather a generic, i.e. language-independent, one. The first step towards this goal is an algorithm that computes generic control dependence. The formalization proposed by Danicic et al. in 2011 appears as a good candidate, since it proposes a generalized version of control dependence defined for arbitrary finite directed graph that unifies previous forms of control dependence. Moreover, their formalization of control dependence comes with an algorithm proved to compute correctly, from a given set, the smallest superset closed under control dependence, which can be viewed as one of the two operations performed to compute a slice. This brings us to the second problem mentioned above. To ensure a high-confidence in the theory and the algorithm, we formalize them in Coq. This formalization helped detect some inaccuracies in one of the proofs. The algorithm is proved correct, but it is quite naive and thus of limited interest for large graphs. We design a new algorithm that optimizes it by transmitting information between iterations. This algorithm is proved correct in Why3 and experimentally clearly outperforms the initial algorithm on random generated graphs.

10.2 Perspectives

In this section, we discuss some research directions that are natural follow-ups of this thesis.

10.2.1 Detection of a Wider Class of Errors

The errors that we address in the justification of slicing for verification are runtime errors. Those are errors determined by the current program state. This kind of errors is appropriate to be modeled by assertions. Other kinds of errors, for instance temporal errors such as use-after-free, cannot be directly handled in the same manner. It would be interesting to explore how to extend our results in this context. The last versions of the Symbiotic tool presented in Section 1.3, Symbiotic 4 and 5 [CVS18], succeed in applying slicing to the detection of memory safety errors, by instrumenting the initial program before slicing it. This could be a track to explore.

10.2.2 A Certified Verification Chain

This thesis provides a formal justification of the use of slicing in verification chains such as SANTE or the one behind the SYMBIOTIC tool (see Section 1.3). This justification is established in a proof assistant, Coq, which ensures a very high confidence in the results. But the other parts of the toolchain are not considered in the formalization. This would be a great improvement if the whole toolchain was proved correct in one unique formalization. As a start point, we could consider the Verasco static analyzer [JLB+15], which is an abstract interpreter formalized in Coq, and tried to interface it with our Coq development justifying the use slicing for verification.

10.2.3 Strong Control Dependence

The second half of this thesis (from Chapter 6) is based on the formalization of weak control dependence on arbitrary finite graphs due to Danicic et al. [DBH+11]. As discussed in Section 6.1.3, weak control dependence is the generalization of non-termination insensitive control dependence. which means in particular that slicing based on this form of control dependence can produce a terminating slice from a non-terminating program. In Section 6.1.3, we mentioned that, in their work, Danicic et al. also propose strong control dependence that generalizes non-termination sensitive control dependence. Contrary to non-termination insensitive control dependence, non-termination sensitive control dependence results in forms of slicing that preserve the non-termination status of the sliced program. In particular, slicing a non-terminating program gives a non-terminating slice. Like for weak control dependence, Danicic et al. also propose an algorithm [DBH+11, Algorithm 65] and prove it correct. Strong control dependence is not addressed by the Coq formalization described in Section 6.6. One research axis would be to formalize it as an extension of the existing Coq development, especially as some concepts are common to both kinds of control dependence and thus are already in the formalization.

10.2.4 Further Experiments

An analysis of the poor performance of the Coq extraction. As discussed in Chapter 9, the Coq extraction of Danicic's algorithm compares poorly with the other implementations. This would be interesting to explore what the origin of this poor performance is. Does this come from the Coq library of graphs that the formalization is based on (cf. Section 6.6.3), from the conversions between the graph representation of this library and that of the OCamlgraph library used to generate the graphs for the experiments, from the Coq formalization itself, or

from the extraction mechanism? We could first try to replace the extraction of the Coq graph library with calls to OCamlgraph. The Coq extraction would then use the same graph library as the other implementations (see Section 9.1.1). If we measured a significant performance improvement, this would mean that the Coq graph library, or at least its extraction into OCaml, is partly responsible of the inefficiency of the Coq extraction.

Experiments on CFGs of real programs. As discussed in Section 9.1.2, the graphs used in the experiments were randomly generated by OCamlgraph. This was suitable for our purpose of experimentally evaluating the complexity of the implementations on arbitrary graphs, but this did not evaluate the algorithms in a realistic context. It would be interesting and complementary to experiment on graphs derived from real programs.

10.2.5 A More Optimized Algorithm for Arbitrary Control Dependence

The optimized algorithm that we have presented in Chapter 7 could itself be optimized, since it is naive in some ways. As suggested in Remark 7.2 and as shown in the experiments (cf. Section 9.3.7), the propagation is certainly the most obvious part of the algorithm that can be improved. In particular, it should be possible in certain cases to detect sooner that some node is weakly deciding. In the current algorithm, we systematically wait for the end of the propagation before detecting weakly deciding nodes. This is because some information is unsure while the propagation is not finished. If we better understood which pieces of information are unsure and which are sure, we could in some cases detect a weakly deciding node directly during the propagation and not at its end.

Other ideas how to improve our algorithm could also be taken from the slicing algorithms proposed by Amtoft et al. in their works on slicing of extended finite state machines [AAC13] and probabilistic programs [AB16]. Their computation of control dependence also detects weakly deciding nodes, but is based on breadth-first search while ours is more depth-first search-based. We could compare our approach with theirs and determine whether they can be mixed.

Another improvement track could be motivated by the integration of our algorithm in a complete slicing algorithm that we discuss in the next subsection. A slicing algorithm would typically interleave the additions of nodes due to control dependence and those due to data dependence. In this thesis, we have shown how to speed up the computation of control dependence thanks to a persistent labeling. But this labeling is only persistent during one step of computation of control dependence. After some nodes have been added due to data dependence, we start a new step of computation of control dependence. *A priori*, a new labeling has to be

used. One goal would be to make the labeling persistent during the whole slicing algorithm. For that, we should describe how to manage in the labeling the nodes added due to data dependence. From the point of view of control dependence, the nodes inserted due to data dependence are nearly arbitrary. The algorithm that we propose in this thesis correctly updates the labeling when detecting a new node because this node is weakly deciding. It is not clear how to modify the algorithm so that the labeling is correctly updated even when the node that is added is not weakly deciding.

10.2.6 A Certified Generic Slicer

Our long-term goal is obviously the formalization of a certified generic slicer. There is still a lot of work to be done about control dependence, but the next step towards this generic slicer is a formalization of generic data dependence. The first question that we should answer is whether we can really formalize a generic form of data dependence or at least what level of genericity we could aim for, since formalizing data dependence seems to imply some formalization of data and memory that could be difficult to model fully generically. For the mechanized aspect of the problem, we could be inspired by the formalizations in Coq of dataflow algorithms, such as that found in the CompCert compiler [Ler09].

Bibliography

[AAC13] Torben Amtoft, Kelly Androutsopoulos, and David Clark. Correctness of slicing finite state machines. Technical Report RN/13/22, University College London, December 2013.

[AB16] Torben Amtoft and Anindya Banerjee. A theory of slicing for probabilistic control flow graphs. In *FoSSaCS*, volume 9634 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2016.

[ADS93] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Softw., Pract. Exper.*, 23(6):589–616, 1993.

[AH03] Matthew Allen and Susan Horwitz. Slicing Java programs that throw and catch exceptions. In *PEPM 2003*, pages 44–54, 2003.

[AHJR14] Min Aung, Susan Horwitz, Richard Joiner, and Thomas W. Reps. Specialization slicing. *ACM Trans. Program. Lang. Syst.*, 36(2):5:1–5:67, 2014.

[All70] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[Alt16] The Alt-Ergo automated theorem prover, v1.30, 2016. http://alt-ergo.lri.fr/.

[Amt08] Torben Amtoft. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Inf. Process. Lett.*, 106(2):45–51, 2008.

[ART03] Paul Anderson, Thomas W. Reps, and Tim Teitelbaum. Design and implementation of a fine-grained software inspection tool. *IEEE Trans. Software Eng.*, 29(8):721–733, 2003.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[BBD+10] Richard W. Barraclough, David Binkley, Sebastian Danicic, Mark Harman, Robert M. Hierons, Akos Kiss, Mike Laurence, and Lahcen Ouarbya. A trajectory-based strict semantics for program slicing. *Theor. Comp. Sci.*, 411(11–13):1372–1386, 2010.

[BC85] Jean-Francois Bergeretti and Bernard Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

[BCCL08] François Bobot, Sylvain Conchon, Evelyne Contejean, and Stéphane Lescuyer. Implementing polymorphism in SMT solvers. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, pages 1–5. ACM, 2008.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[BdCHP12] José Bernardo Barros, Daniela Carneiro da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. *Formal Asp. Comput.*, 24(2):217–248, 2012.

[BDP15] Sandrine Blazy, Delphine Demange, and David Pichardie. Validating dominator trees for a fast, verified dominance test. In *ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2015.

[BFH+16] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, v1.12*, 2016.

[BG96] David W. Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[BGH+14] David W. Binkley, Nicolas Gold, Mark Harman, Syed S. Islam, Jens Krinke, and Shin Yoo. ORBS: language-independent program slicing. In *SIGSOFT FSE*, pages 109–120. ACM, 2014.

[BH93] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *AADEBUG 1993*, 1993.

[BH04] David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[BHR95] David W. Binkley, Susan Horwitz, and Thomas W. Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.

[Bin98] David W. Binkley. The application of program slicing to regression testing. *Information & Software Technology*, 40(11-12):583–594, 1998.

[BMP15] Sandrine Blazy, André Maroneze, and David Pichardie. Verified validation of program slicing. In *CPP 2015*, pages 109–117, 2015.

[BP96] Gianfranco Bilardi and Keshav Pingali. Generalized dominance and control dependence. In *PLDI*, pages 291–300. ACM, 1996.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *the 4th Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, 1977.

[CCK+14] O. Chebaro, P. Cuoq, N. Kosmatov, B. Marre, A. Pacalet, N. Williams, and B. Yakobowski. Behind the scenes in SANTE: a combination of static and dynamic analyses. *Autom. Softw. Eng.*, 21(1):107–143, 2014.

[CCL98] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information & Software Technology*, 40(11-12):595–607, 1998.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CF89] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *PLDI 1989*, 1989.

[CFS07] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ML functors. In *Trends in Functional Programming*, volume 8 of *Trends in Functional Programming*, pages 124–140. Intellect, 2007.

[CH96] Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In *FME*, volume 1051 of *Lecture Notes in Computer Science*, pages 557–575. Springer, 1996.

[CHK01] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

[CKGJ12] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC 2012*, 2012.

[Coq17] The Coq Development Team. The Coq proof assistant, v8.6. http://coq.inria.fr/, 2017.

[CVC17] CVC4, the smt solver, v1.5. http://cvc4.cs.stanford.edu/web/, 2017.

[CVS18] Marek Chalupa, Martina Vitovská, and Jan Strejček. SYMBIOTIC 5: Boosted instrumentation - (competition contribution). In *TACAS (2)*, volume 10806 of *Lecture Notes in Computer Science*, pages 442–446. Springer, 2018.

[DBH+11] Sebastian Danicic, Richard W. Barraclough, Mark Harman, John Howroyd, Ákos Kiss, and Michael R. Laurence. A unifying theory of control dependence and its application to arbitrary program structures. *Theor. Comput. Sci.*, 412(49):6809–6842, 2011.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[DERV15] Catherine Dubois, Sourour Elloumi, Benoit Robillard, and Clément Vincent. Graphes et couplages en Coq. In *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, 2015. In French.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[E17] The E Theorem Prover, v2.0. http://eprover.org, 2017.

[FGMT13] Wojciech Fraczak, Loukas Georgiadis, Andrew Miller, and Robert E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms*, 23:2 – 20, 2013. 23rd International Workshop on Combinatorial Algorithms (IWOCA 2012).

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[FRT95] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *POPL*, pages 379–392. ACM Press, 1995.

[FSF07] Free Software Foundation. GNU General Public License, version 3. http://www.gnu.org/licenses/gpl.html, June 2007.

[GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Software Eng.*, 17(8):751–761, 1991.

[GM03] Roberto Giacobazzi and Isabella Mastroeni. Non-standard semantics for program slicing. *Higher-Order and Symbolic Computation*, 16(4):297–339, 2003.

[Gon08] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[GTW06] Loukas Georgiadis, Robert Endre Tarjan, and Renato Fonseca F. Werneck. Finding dominators in practice. *J. Graph Algorithms Appl.*, 10(1):69–94, 2006.

[GTXT11] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. DyTa: dynamic symbolic execution guided with static verification results. In *the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 992–994. ACM, 2011.

[HBD03] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, 2003.

[HCD+99] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *SAS*, volume 1694 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1999.

[HD95] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Softw. Test., Verif. Reliab.*, 5(3):143–162, 1995.

[HHD99] Robert M. Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Softw. Test., Verif. Reliab.*, 9(4):233–262, 1999.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):567–580, 1969.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *PLDI 1988*, 1988.

[HSD96] Mark Harman, Dan Simpson, and Sebastian Danicic. Slicing programs in the presence of errors. *Formal Aspects of Computing*, 8(4):490–497, 1996.

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *POPL*, pages 247–259. ACM, 2015.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[KL88] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[KNNI02] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7(1):49–76, 2002.

[KR98] Bogdan Korel and Juergen Rilling. Program slicing in understanding of large programs. In *IWPC*, pages 145–152. IEEE Computer Society, 1998.

[Kri05] Jens Krinke. Program slicing. In *Handbook Of Software Engineering And Knowledge Engineering: Vol 3: Recent Advances*, pages 307–332. World Scientific, 2005.

[LCYK01] Wan Kwon Lee, In Sang Chung, Gwang Sik Yoon, and Yong Rae Kwon. Specification-based program slicing and its applications. *Journal of Systems Architecture*, 47(5):427–443, 2001.

[Léc16] Jean-Christophe Léchenet. Formalization of relaxed slicing, 2016. http://perso.ecp.fr/~lechenetjc/slicing/.

[Léc18] Jean-Christophe Léchenet. Formalization of weak control dependence, 2018. http://perso.ecp.fr/~lechenetjc/control/.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[Lio96] J. L. Lions. Ariane 5 Flight 501 Failure – Report by the Inquiry Board. Technical report, European Space Agency, July 1996.

[LKL16a] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Coq a dit : fromage tranché ne peut cacher ses trous. In *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*, 2016. In French.

[LKL16b] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Cut branches before looking for bugs: Sound verification on relaxed slices. In *FASE'16 (Part of ETAPS'16)*, pages 179–196, 2016.

[LKL18a] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Cut branches before looking for bugs: certifiably sound verification on relaxed slices. *Formal Aspects of Computing*, 30(1):107–131, Jan 2018.

[LKL18b] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Why3 a dit : gardez le contrôle en toute situation. In *Vingt-neuvièmes Journées Francophones des Langages Applicatifs (JFLA 2018)*, pages 219–225, 2018. In French.

[LKL18c] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Fast Computation of Arbitrary Control Dependencies. In Alessandra Russo and Andy Schürr, editors, *FASE'18 (Part of ETAPS'18)*, pages 207–224, Cham, 2018. Springer International Publishing.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[MZ17] Isabella Mastroeni and Damiano Zanardini. Abstract program slicing: An abstract interpretation-based approach to program slicing. *ACM Trans. Comput. Log.*, 18(1):7:1–7:58, 2017.

[Nes09] Härmel Nestra. Transfinite semantics in the form of greatest fixpoint. *J. Log. Algebr. Program.*, 78(7):573–592, 2009.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1984)*, pages 177–184. ACM Press, 1984.

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Software Eng.*, 16(9):965–979, 1990.

[PCG+15] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations 3.2, 2015. http://www.cis.upenn.edu/~bcpierce/sf/sf-3.2/index.html.

[PKJ06] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. Bug classification using program slicing metrics. In *SCAM*, pages 31–42. IEEE Computer Society, 2006.

[RAB+07] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[RB89] Thomas W. Reps and T. Bricker. Illustrating interference in interfering versions of programs. In *SCM*, pages 46–55. ACM Press, 1989.

[RH07] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using indus and kaveri. *International Journal on Software Tools for Technology Transfer*, 9(5):489–504, Oct 2007.

[Riv05] Xavier Rival. Understanding the origin of alarms in astrée. In *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 303–319. Springer, 2005.

[RY89] Thomas W. Reps and Wuu Yang. The semantics of program slicing and program integration. In *TAPSOFT 1989*, 1989.

[Sch13] Stephan Schulz. System description: E 1.8. In *LPAR*, volume 8312 of *Lecture Notes in Computer Science*, pages 735–743. Springer, 2013.

[Sil12] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12, 2012.

[SST12] Jiří Slabý, Jan Strejček, and Marek Trtík. Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution. In *FMICS*, volume 7437 of *Lecture Notes in Computer Science*, pages 207–221. Springer, 2012.

[SST13] Jiri Slaby, Jan Strejček, and Marek Trtík. Symbiotic: Synergy of instrumentation, slicing, and symbolic execution - (competition contribution). In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 630–632. Springer, 2013.

[Tip95] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *PLDI*, pages 107–119. ACM, 1991.

[Was11] Daniel Wasserrab. *From formal semantics to verified slicing: a modular framework with applications in language based security.* PhD thesis, Karlsruhe Inst. of Techn., 2011.

[Wei81] Mark Weiser. Program slicing. In *ICSE 1981*, 1981.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[Wei84] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[Why18] Why3, a tool for deductive program verification, GNU LGPL 2.1, development version. http://why3.lri.fr, January 2018.

[WLS09] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On pdg-based noninterference and its modular proof. In *PLAS*, pages 31–44. ACM, 2009.

[WZ07] Martin P. Ward and Hussein Zedan. Slicing as a program transformation. *ACM Trans. Program. Lang. Syst.*, 29(2):7, 2007.

[XQZ+05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[Z16] The Z3 Theorem Prover, v4.5.0. https://github.com/Z3Prover/z3, 2016.

Index

$(v \to T, T')$, 27, 38, 71
$\langle (l, \sigma) \rangle \downarrow L$, 29, 60, 87
$\leq_q$, 49, 80
$\leq_s$, 53
$\circlearrowleft$, 81
$\oplus$, 26, 28, 38, 43, 71
$\lightning$, 81
$\xrightarrow{V'-disjoint}$, 30, 127
$\xrightarrow{V'-path}$, 30, 121
$\xrightarrow{\mathcal{D}_a}$, 77
$\xrightarrow{\mathcal{D}_c}$, 42, 75
$\xrightarrow{\mathcal{D}_d}$, 46, 77
$\xrightarrow{path}$, 29, 120
—, 81

assertion dependence, 77

CFG, 112
`confirm`, 165
    annotated version, 183
control dependence
    on CFGs, 114, 115, 117
    on structured programs, 42, 75, 112
Coq, 11

$\mathcal{D}_a$, 77
Danicic's algorithm, 139
    optimization 1, 140, 212
    optimization 2, 140, 212
data dependence, 46, 77
$\mathcal{D}_c$, 42, 75
$\mathcal{D}_d$, 46, 77
$\mathsf{def}(l)$, 45, 76

$\mathcal{E}[\![e]\!]\sigma$, 38, 72

finite syntactic path, 28, 44, 76
$\mathcal{F}_L(p)$, 51

$L(p)$, 35
$LS_\sigma(T)$, 27, 38, 71

`main`, 167
    annotated version, 185

observable
    set, 30, 121
    vertex, 121
$\mathsf{obs}_G(u, V')$, 30, 121

$\mathcal{P}(p)$, 44, 76
path, 29, 120
post-dominator, 113, 116
projection
    of a state, 28, 60, 87
    of a trajectory, 29, 60, 87
$\mathsf{Proj}_L(T)$, 29, 60, 87
`propagate`, 164
    annotated body, 174
    contract, 172

quotient, 49, 80

R^*, 47
R^{-1}, 47
reachable nodes, set of, 30, 120

$\mathsf{R}_G(V')$, 30, 120
$\mathsf{ref}(l)$, 45, 76
R^n, 47

$\sigma{\downarrow}V$, 28, 60, 87
$\sigma[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n]$, 73
$\sigma[x \leftarrow v]$, 26, 73
slice, 56, 80
slice set, 47, 78
soundness theorem of slicing
    classic property, 61
    relaxed property, 87
state, 25, 36, 71
sub-program, 53

$T^{(n)}$, 27, 62, 88
$\mathcal{T}[\![p]\!]\sigma$, 39, 72
trajectory, 26, 39, 72

$\mathsf{used}(l)$, 46, 76

V'-critical edge, 137
V'-disjoint path, 30, 127
V'-path, 30, 121
V'-weakly committing vertex, 30, 122
V'-weakly deciding vertex, 30, 125
$\mathsf{vars}(e)$, 35

$\mathsf{WCC}_G(V')$, 31, 134
$\mathsf{WC}_G(V')$, 30, 122
$\mathsf{WD}_G(V')$, 30, 125
weak control-closure, 31, 134
weakly control-closed set, 123
WHILE language
    semantics, 39
    syntax, 34
WHILE language with errors
    semantics, 72
    syntax, 67
Why3, 18

ÉCOLE DOCTORALE
INTERFACES
Approches Interdisciplinaires : Fondements, Applications et Innovation

Titre : Algorithmes certifiés pour la simplification syntaxique de programmes

Mots-clefs : Slicing, vérification formelle, dépendances de contrôle, théorie des graphes, Coq, Why3

La simplification syntaxique, ou slicing, est une technique permettant d'extraire, à partir d'un programme et d'un critère consistant en une ou plusieurs instructions de ce programme, un programme plus simple, appelé slice, ayant le même comportement que le programme initial vis-à-vis de ce critère.

Les méthodes d'analyse de code permettent d'établir les propriétés d'un programme. Ces méthodes sont souvent coûteuses, et leur complexité augmente rapidement avec la taille du code. Il serait donc souhaitable d'appliquer ces techniques sur des slices plutôt que sur le programme initial, mais cela nécessite de pouvoir justifier théoriquement l'interprétation des résultats obtenus sur les slices.

Cette thèse apporte cette justification pour le cas de la recherche d'erreurs à l'exécution. Dans ce cadre, deux questions se posent. Si une erreur est détectée dans une slice, cela veut-il dire qu'elle se déclenchera aussi dans le programme initial ? Et inversement, si l'absence d'erreurs est prouvée dans une slice, cela veut-il dire que le programme initial en est lui aussi exempt ? Nous modélisons ce problème sur un mini-langage impératif représentatif, autorisant les erreurs et la non-terminaison, et montrons le lien entre la sémantique du programme initial et la sémantique de sa slice, ce qui nous permet de répondre aux deux questions précédentes.

Pour généraliser ces résultats, nous nous intéressons à la première brique d'un slicer indépendant du langage : le calcul générique des dépendances de contrôle. Nous formalisons une théorie élégante de dépendances de contrôle sur des graphes orientés finis arbitraires prise dans la littérature et améliorons l'algorithme de calcul proposé.

Pour garantir un maximum de confiance dans les résultats, tous ces travaux sont prouvés dans l'assistant de preuve Coq ou dans l'outil de preuve Why3.

Title: Certified algorithms for program slicing

Keywords: Program slicing, formal verification, control dependence, graph theory, Coq, Why3

Program slicing is a technique that extracts, given a program and a criterion that is one or several instructions in this program, a simpler program, called a slice, that has the same behavior as the initial program with respect to the criterion.

Program analysis techniques focus on establishing the properties of a program. These techniques are costly, and their complexity increases with the size of the program. Therefore, it would be interesting to apply these techniques on slices rather than the initial program, but it requires theoretical foundations to interpret the results obtained on the slices.

This thesis provides this justification for runtime error detection. In this context, two questions arise. If an error is detected in the slice, does this mean that it can also be triggered in the initial program? On the contrary, if the slice is proved to be error-free, does this mean that the initial program is error-free too? We model this problem using a small representative imperative language containing errors and non-termination, and establish the link between the semantics of the initial program and of its slice, which allows to give a precise answer to the two questions raised above.

To apply these results in a more general context, we focus on the first step towards a language-independent slicer: an algorithm computing control dependence. We formalize an elegant theory of control dependence on arbitrary finite directed graphs taken from the literature and improve the proposed algorithm.

To ensure a high confidence in the results, we prove them in the Coq proof assistant or in the Why3 proof plateform.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France